RISC-V Y SU ECOSISTEMA HARDWARE-SOFTWARE

RISC-V HARDWARE-SOFTWARE ECOSYSTEM

ESTÍBALIZ BUSTO PÉREZ DE MENDIGUREN

GRADO EN INGENIERÍA INFORMÁTICA. FACULTAD DE INFORMÁTICA

UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin Grado en Ingeniería Informática

Febrero 2021

Directores:

Fernando Castro Rodriguez

Katzalin Olcoz Herrero

Autorización de difusión

Estíbaliz Busto Pérez de Mendiguren

2020/2021

La abajo firmante, matriculada en el Grado de Ingeniería Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Grado: "RISC-V Y SU ECOSISTEMA HARDWARE-SOFTWARE", bajo la dirección de Fernando Castro Rodriguez y Katzalin Olcoz Herrero en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Dedicatoria

Invertir el tiempo en conocimiento

es una buena inversión.

Agradecimientos

Quiero expresar mi agradecimiento a todas y cada una de las personas que de forma directa o indirecta han hecho posible el presente Trabajo Fin de Grado.

En primer lugar, a mis directores, Fernando y Katzalin, por el apoyo recibido ya que sin su ayuda, paciencia y conocimiento este Trabajo Fin de Grado no hubiese sido posible.

Asimismo, gracias a todo el profesorado que he tenido durante estos años estudiando el Grado de Ingeniería Informática en la UCM, que con su dedicación me han enseñado y ayudado en mi día a día. Todo ello, sin olvidarme de aquellos que me han ofrecido ayuda durante el confinamiento producido por la emergencia sanitaria, puesto que sin su conocimiento e interés este TFG no se hubiera llevado a cabo.

A mi familia, por todo lo que han enseñado y orientado, por anticiparse y por darme la oportunidad de estudiar donde lo he hecho. Gracias a su esfuerzo, paciencia, ánimo y apoyo incondicional en todo momento a lo largo de esta etapa porque sin ella no hubiera podido superar las dificultades encontradas; y, en particular, a mi hermano, aunque sea un “trotamundos”, siempre ha estado disponible en todo momento allá donde estuviera.

Finalmente, a todas las personas que he conocido durante estos años porque han merecido la pena.

Por todo ello, muchas gracias.

Resumen

Desde el año 2010, se lleva investigando en una Arquitectura (ISA) de hardware libre basada en el diseño de tipo RISC, llamada RISC-V. RISC-V destaca por ser una arquitectura abierta, que pretende convertirse en universal, es decir, persistir en el tiempo.

En este Trabajo Fin de Grado se llevará a cabo un estudio de la arquitectura RISC-V, sus características principales y su ecosistema Software y Hardware. Para el estudio del ecosistema Software se analizarán diferentes simuladores orientados al ámbito educativo y herramientas software, como depuradores y entornos de desarrollo. Para el ecosistema Hardware se presentarán varias empresas que diseñan núcleos que utilizan la arquitectura RISC-V, placas y SoCs.

Por último, se aplicarán los conocimientos adquiridos para analizar la placa SparkFun Red-V Thing Plus realizando un tutorial básico de cómo hacer un programa en C y otro en ensamblador usando dos IDE distintos (PlatformIO y Freedom Studio de SiFive).

Palabras clave

Arquitectura RISC-V, ISA, Software, Hardware, SparkFun Red-V Thing Plus.

Abstract

Since 2010, research has been carried out on a free hardware Architecture (ISA) based on the RISC-type design, called RISC-V. RISC-V stands out for being an open architecture, which aims to become universal, that is, to persist over time.

In this Final Degree Project, a study of the RISC-V architecture, characteristics and its Software and Hardware ecosystem will be carried out. For the study of the Software ecosystem, different simulators oriented to the educational field and software tools such as development environments will be analyzed, as well as the operating systems. For the Hardware ecosystem, several companies that design cores using the RISC-V architecture will be presented.

Finally, the acquired knowledge will be applied to analyze the SparkFun Red-V Thing Plus board doing a basic tutorial on how to make a program in C and assembler program languages, using two different IDE (PlatformIO and Freedom Studio from SiFive).

Keywords

RISC-V architecture, ISA, Software, Hardware, SparkFun Red-V Thing Plus

Índice de contenidos

Índice de tablas

Índice de figuras

Capítulo 1 - Introducción

RISC-V, pronunciado *"risk-five"*, es una arquitectura (ISA) de hardware libre basada en un diseño de tipo RISC (conjunto de instrucciones reducido).

El proyecto comenzó en 2010 en la Universidad de Berkeley (California). Aunque estuvo orientado a la investigación y la docencia, tuvo una considerable participación externa a la Universidad procedente del sector industrial. El profesor Krste Asanović y los estudiantes de posgrado Yunsup Lee y Andrew Waterman comenzaron la instrucción RISC-V, teniendo como director al Profesor David Patterson [1]. Sus diseñadores pretenden convertirla en un estándar de arquitectura abierta para diversas aplicaciones, de manera gratuita, evitando así el pago de licencias.

La arquitectura RISC-V nos permitirá tener programas y sistemas más eficaces y estables, sin tener costes adicionales. Por ello, resulta interesante su estudio para, así, poder descubrir sus beneficios.

1.1 Arquitecturas CISC y RISC

A mediados del siglo XX, los computadores se diseñaban de forma aislada unos de otros, es decir, sus repertorios de instrucciones eran independientes, con lo que un programa escrito para un computador no se podía ejecutar en otro. Años más tarde, investigadores de IBM (1964) consiguieron ampliar la compatibilidad del software en diferentes máquinas y crearon la arquitectura CISC (*Complex Instruction Set Computer* / Conjunto Complejo de Instrucciones de Computadores), un modelo de ISA complejo. Este tipo de arquitectura es utilizada por microprocesadores x86, la minicomputadora VAX y la computadora IBM S/360.

En la década de los 70, los científicos de IBM empezaron a diseñar una alternativa que posteriormente se introdujo en el mercado, RISC. El IBM 801 que empezó a crearse en 1975, fue diseñado por John Cocke y es considerado el primer procesador RISC de la historia.

Los investigadores de IBM (1975) diseñaron otro tipo de procesadores con la arquitectura RISC (*Reduced Instruction Set Computer* / Conjunto Reducido de Instrucciones de Computadores). RISC es un tipo de diseño de CPU utilizado principalmente en microprocesadores o microcontroladores con instrucciones de tamaño fijo. Los primeros diseños realizados fueron Berkerley RISC-I (*David A. Patterson),* Standfor MIPS (*John L. Hennessy)* y la minicomputadora IBM 801 *(Cocke).*

El objetivo de diseñar máquinas RISC es posibilitar la segmentación y el paralelismo en la ejecución de instrucciones y reducir los accesos a memoria. Este tipo de arquitectura es utilizada por el PowerPC y arquitecturas como ARM, SPARC y Alpha.

A continuación, en la Tabla 1-1, se muestran las principales características comparativas entre CISC y RISC [3]:

CISC **(Complex Instruction Set Computer)**	RISC **(Reduced Instruction Set Computer)**
Arquitectura RM y MM.	Arquitectura RR (carga/almacenamiento).
Diseñadas sin tener en cuenta la verdadera demanda de los programas.	Diseñadas a partir de las mediciones practicadas en los programas a partir de los años 80.
Objetivo: dar soporte arquitectónico a las funciones requeridas por los LANs.	Objetivo: dar soporte eficiente a los casos frecuentes.
Muchas operaciones básicas y tipos de direccionamiento complejos.	Pocas operaciones básicas y tipos de direccionamiento simples.
Instrucciones largas y complejas con formatos muy diversos: decodificación compleja (lenta).	Instrucciones de formato simple (tamaño fijo): decodificación simple (rápida).
Pocas instrucciones por programa: elevado número de ciclos por instrucción (CPI).	Muchas instrucciones por programa: reducido número de ciclos por instrucción (CPI).
Muchos tipos de datos: interfaz con memoria compleja.	Sólo los tipos de datos básicos: interfaz con memoria sencilla.
Número limitado de registros de propósito general: mucho almacenamiento temporal en memoria.	Número elevado de registros: uso eficiente por el compilador para almacenamiento temporal.
Baja ortogonalidad en las instrucciones: muchas excepciones para el compilador.	Alto grado de ortogonalidad en las instrucciones: mucha regularidad para el compilador.

Tabla 1-1. Características comparativas entre CISC y RISC.
Fuente: Ortiz, J. J. (20 de septiembre de 2018). Grado en Ingeniería del Software. Obtenido de http://www.fdi.ucm.es/profesor/jjruz/EC-IS/Temas%02-Arquitectura%20del%20procesador.pdf

1.2 RISC-V

RISC-V es un ISA de hardware libre, de código abierto bajo licencia BSD *(Berkeley Software Distribution)*, basada en el diseño de tipo RISC (*Reduced Instruction Set Computer*). La arquitectura RISC-V surgió como apoyo a las investigaciones en arquitecturas hardware y para facilitar el aprendizaje de ellas. Al ser una arquitectura libre, nos permite diseñar y fabricar chips utilizando dicha arquitectura para diversas aplicaciones, de manera gratuita, evitando el pago de licencias [2]. Esto le convierte a RISC-V en una arquitectura interesante, tanto a nivel docente como comercial, frente a las arquitecturas nombradas anteriormente.

RISC-V surgió a raíz de la realización de varios proyectos académicos de diseño de computadores, orientados principalmente a la investigación y la docencia. El proyecto comenzó en 2010 en la Universidad de Berkeley (California) con el propósito de convertirla en un estándar de arquitectura abierta para diversas aplicaciones. El profesor Krste Asanović encontró muchos usos para un sistema informático de código abierto. En 2010, desarrolló y publicó un "proyecto corto de tres meses durante el verano" con el objetivo de ayudar a los usuarios académicos e industriales. David Patterson, su gran colaborador, es considerado como uno de los pioneros en RISC. En 1980, lideró cuatro generaciones de proyectos RISC (de ahí el nombre RISC-V); autor de cinco libros, dos de los libros fueron escritos con el profesor John Hennessy y tratan sobre la arquitectura de computadores. Hennessy se convirtió en miembro de la Universidad de Stanford en 1977. En 1984, fundó MIPS Computer Systems Inc para comercializar los procesadores RISC sobre los que había estado investigando. Andrew Waterman codiseñó el ISA RISC-V, es uno de los principales contribuyentes al generador de chips Rocket de código abierto basado en RISC-V y el primer microprocesador RISC-V. Además, es el cofundador de la empresa SiFive. SiFive fue fundada por los creadores de la arquitectura RISC-V para proporcionar chips a la medida basados en RISC-V.

RISC-V se estructura en torno a su Fundación: “RISC-V Foundation”. Con sede en Suiza, actualmente tiene más de 300 miembros (empresas, entidades, organismos…) entre los que se incluyen firmas como Google, Qualcomm, NVIDIA, Samsung, Western Digital, IBM o Micron; pero también otras de fuera de Estados Unidos, como el fabricante de chips europeo NXP Semiconductors, el gigante del comercio electrónico Grupo Alibaba y Huawei Technologies o universidades como la Universidad Complutense de Madrid.

Una de las novedades que aporta RISC-V es su diseño modular y extensible. El núcleo fundamental del ISA es el RV32I. El RV32I nunca cambia y proporciona estabilidad para desarrolladores de compiladores, sistemas operativos y programadores de lenguaje ensamblador. Su modularidad se produce gracias a extensiones opcionales estándar que el hardware puede incorporar según la necesidad de la aplicación que se vaya a desarrollar. Además, dicha modularidad posibilita implementaciones de RISC-V muy pequeñas y de bajo consumo energético.

Aunque no es la primera ISA de arquitectura abierta, es significativa porque está diseñada para ser útil y eficaz en una amplia gama de dispositivos, desde los más pequeños hasta los más rápidos. RISC-V es un ISA reciente y abierto, que aprovecha las buenas ideas de las arquitecturas anteriores, evitando sus errores.

En RISC-V todas las operaciones se realizan entre registros, salvo load/store que sí utilizan accesos a memoria. RV32I tiene 32 registros de propósito general que contienen el estado del procesador, los datos que se operan en cada momento y la información de mantenimiento. El banco de registros es capaz de minimizar la necesidad de acceder a la memoria externa para muchas tareas básicas de la CPU, lo que reduce el uso de energía y aumenta la velocidad.

El objetivo fundamental de RISC-V es convertirse en un ISA universal por ser un ISA abierto, centrado en mantener la estabilidad de RISC-V de forma cuidadosa y lenta, basada principalmente en razones técnicas.

A continuación, en la Tabla 1-2 se muestran las características técnicas [4] de RISC-V:

Diseñador	Universidad de California, Berkeley.
Gestión	RISC-V Foundation.
Licencia	BSD (abierta y gratuita).
Tamaño de palabra	32 bits, 64 bits, 128 bits (con extensiones SIMD o vectoriales).
Tipo	RISC, load/store.
Núcleo	RV32I, RV64I, RV128I (no cambia, tiene extensiones modulares).
Extensiones modulares	M (instrucciones para multiplicar y dividir). A (operaciones atómicas). F (aritmética de punto flotante de precisión simple). D (aritmética punto flotante de precisión doble). Q (aritmética punto flotante de precisión cuádruple). L (aritmética punto flotante decimal). C (instrucciones comprimidas). B (manipulación de bits). J (Lenguajes traducidos dinámicamente). P (para instrucciones SIMD). V (vectoriales). E (aplicaciones de sistemas empotrados) y G (conjunto M, A, F y D).
Tipos de registros	De propósito general (16 o 32, registro x0: siempre a valor 0x0).
Codificación	Variable.
Branching	Comparación y saltos.
Endianness	Little-endian.
ISA modular	Se pueden ir agregando módulos a partir del ISA fijo.

Tabla 1-2. Características técnicas de RISC-V.
Fuente: (13 de junio de 2019). Architecnologia. Obtenido de https://architecnologia.es/risc-v-introduccion-a-la-isa-parte-2

En el Capítulo 2, se llevará a cabo un análisis más profundo de la arquitectura RISC-V: el estudio del repertorio de instrucciones (el núcleo fundamental del ISA RV32I), el formato de instrucciones, el banco de registros, los modos de direccionamiento y el lenguaje ensamblador.

1.3 Motivación

Desde mi infancia siempre he mostrado especial interés por aparatos electrónicos, herramientas, cables... Aunque la actual crisis sanitaria me lo ha complicado bastante, innovarme y superarme son algunos de mis retos. Por eso, no tengo ninguna duda que el conjunto de instrucciones de RISC-V diseñadas pensando en implementaciones rápidas y de bajo consumo para el mundo real favorecen una mayor innovación debido a una competición de mercado abierto por muchos diseñadores nuevos.

Mi innata curiosidad por lo desconocido, unida a mi continua formación académica proporcionada por el Grado de Ingeniería Informática, ha hecho despertar mi interés por el mundo del Hardware. Además, mi interés por la asignatura de Arquitectura de Computadores aumentó dicho interés ante la propuesta de mis tutores. Por ello, estoy convencida que este Trabajo de Fin de Grado me posibilita conocer y profundizar en ello.

Para finalizar, creo firmemente que debemos aprovechar la arquitectura RISC-V de hardware libre, esto permite que cualquier persona con conocimientos pueda participar en su desarrollo, conseguir mejorar el futuro sin costes adicionales, más en estos momentos tan difíciles a los que nos enfrentamos.

1.4 Objetivos

El objetivo principal de este Trabajo de Fin de Grado es realizar un estudio de la arquitectura RISC-V y su ecosistema hardware y software, a partir de toda la información y documentación obtenida para tal propósito. Por otra parte, se llevará a cabo el análisis y estudio de la placa *SparkFun Red-V Thing Plus* que utiliza la arquitectura RISC-V.

El presente documento se estructura en seis capítulos descritos a continuación:

Capítulo 1 - “Introducción”. Comienza con una brevísima historia de las arquitecturas RISC y CISC antes de centrarnos en la arquitectura RISC-V; la motivación que llevó a la realización de este proyecto, los objetivos junto con la estructura del documento y el plan de trabajo son los apartados que completan dicho capítulo.

Capítulo 2 - “Estudio del repertorio de instrucciones”. Se analizarán los aspectos más esenciales de la arquitectura RISC-V como el formato de instrucciones, los modos de direccionamiento, el banco de registros, los modos de la CPU y el lenguaje ensamblador.

Capítulo 3 - “Estudio del ecosistema Software”. Se explicarán algunos de los simuladores, toolchains, depuradores y sistemas operativos que soportan la arquitectura RISC-V.

Capítulo 4 - “Estudio del ecosistema Hardware”. Se abordarán algunos de los cores, SoC y placas que implementan este juego de instrucciones.

Capítulo 5 - “Casos de uso”. Se analizará la placa SparkFun Red-V Thing Plus, realizando un tutorial básico de cómo hacer un programa en C y otro en ensamblador usando dos IDE distintos: PlatformIO en Visual Studio Code y Freedom Studio de SiFive.

Capítulo 6 - “Conclusiones”. Se analizará tanto el objetivo del trabajo, como las conclusiones técnicas que derivan del mismo, además de exponer las dificultades tenidas y proponer ideas para posibles trabajos futuros.

Para finalizar, en los apéndices se desarrollarán en mayor profundidad algunos temas que, aunque no se consideren imprescindibles para el seguimiento del presente proyecto, se consideran de interés para su lector y cuyo objetivo es ampliar los contenidos del documento principal.

En definitiva, el objetivo principal del presente Trabajo de Fin de Grado es realizar una guía basada en una recopilación de información obtenida a través del estudio e investigación de diferentes guías, manuales, documentos, foros … y así, poder descubrir los beneficios que nos puede aportar una arquitectura nueva como es RISC-V.

1.5 Plan de trabajo

Desde el primer momento el contacto con los tutores del presente TFG se ha realizado por medio de e-mails, reuniones presenciales o videollamadas.

En la fase inicial, junto con los tutores, se definió el proyecto de investigación, el método de trabajo y la documentación básica tenida en cuenta para la realización de dicho proyecto.

Seguidamente, se procedió a la planificación del proyecto mediante la búsqueda, selección y organización de la información. Hubo que definir los objetivos, especificar las tareas a realizar y plasmar la información conseguida en los capítulos definidos en el apartado 1.4.

A medida que se iban adquiriendo conocimientos del tema investigado, se tomaban los apuntes oportunos para ir redactando poco a poco la memoria (revisable en todo momento para sus modificaciones pertinentes).

Posteriormente, se llevó a cabo el estudio y análisis de las placas Sipeed Longan Nano y Arty A7. Pese al tiempo invertido en ellas, no se consiguió ningún resultado debido a dificultades provocadas por:

a) El confinamiento provocado por la emergencia sanitaria del COVID-19 tuvo una fuerte repercusión a la hora de no poder contar con los medios presenciales, ni materiales necesarios desde mi domicilio donde tuve que pasar dicho confinamiento.

b) Los reiterados intentos para el funcionamiento de las placas no tuvieron el éxito deseado, tal y como se hubiera esperado:

    - Se hicieron particiones en el equipo instalando Ubuntu por falta de espacio en el disco duro, llegando a utilizar otros equipos para comprobar su funcionamiento.
    - Se realizaron pruebas con un voltímetro para medir la corriente en los distintos puntos de la placa para comprobar si estaban o no defectuosas.
    - Falta de documentación en español, la mayoría en chino, así como la falta de respuestas en los foros destinados a este propósito.

Finalmente, se elaboró el proyecto mediante la redacción de la presente memoria y su posterior presentación oral.

Chapter 1 - Introduction

RISC-V, pronounced "risk-five", is a free hardware instruction set architecture (ISA) based on a RISC (reduced instruction set) type design.

The project began in 2010 at the University of Berkeley (California). Although it was oriented to research and teaching, it had considerable participation outside the University from the industrial sector. Professor Krste Asanović and graduate students Yunsup Lee and Andrew Waterman began RISC-V instruction, led by Professor David Patterson [1]. Its designers intend to turn it into an open architecture standard for various applications, free of charge, thus avoiding the payment of licenses.

RISC-V architecture will allow us to have more efficient and stable programs and systems, without having additional costs. Therefore, its study is interesting to discover its benefits.

1.1 CISC and RISC Architectures

In the middle of the 20th century, computers were designed in isolation from each other, that is, their instruction sets were independent, so that a program written for one computer could not be executed on another. Later, IBM researchers (1964) managed to expand the compatibility of the software on different machines and created the CISC *(Complex Instruction Set Computer)* architecture, a complex ISA model. This type of architecture is used by x86 microprocessors, the VAX minicomputer, and the IBM S / 360 computer.

In the 1970s, IBM scientists designed an alternative that was later introduced to the market, RISC. The IBM 801, which began to be created in 1975, was designed by John Cocke and it is considered the first RISC processor in history.

IBM researchers (1975) designed other types of processors with the RISC architecture *(Reduced Instruction Set Computer)*. RISC is a type of CPU design used primarily in microprocessors or microcontrollers with fixed-size instructions. The first designs made were Berkerley RISC-I (David A. Patterson), Standfor MIPS (John L. Hennessy) and the IBM 801 minicomputer (Cocke).

The objective of designing RISC machines is to enable segmentation and parallelism in the execution of instructions by reducing memory accesses. This type of architecture is used by PowerPC and architectures such as ARM, SPARC and Alpha.

Next, in Table 1-1, the main comparative characteristics [3] between CISC and RISC are shown:

CISC (Complex Instruction Set Computer)	RISC (Reduced Instruction Set Computer)
RM and MM architecture.	RR architecture (load / store).
Designed without taking into account the true demand of the programs.	Designed from measurements made in programs from the 80's years.
Objective: to provide architectural support to the functions required by the LANs.	Objective: to give efficient support to frequent cases.
Many basic operations and complex addressing types.	Few basic operations and simple addressing types.
Long and complex instructions with very diverse formats: complex (slow) decoding.	Simple format instructions (fixed size): simple (fast) decoding.
Few instructions per program: high number of cycles per instruction (CPI).	Many instructions per program: low number of cycles per instruction (CPI).
Many types of data: complex memory interface.	Only basic data types: simple memory interface.
Limited number of general purpose registers: a lot of temporary memory storage.	High number of records: efficient use by the compiler for temporary storage.
Low orthogonality in the instructions: many exceptions for the compiler.	High degree of orthogonality in the instructions: a lot of regularity for the compiler.

Table 1-1. Main comparative characteristics between CISC and RISC.
Source: Ortiz, J. J. (September 20, 2018). Grado en Ingeniería del Software. Obtained from http://www.fdi.ucm.es/profesor/jjruz/EC-IS/Temas%02-Arquitectura%20del%20procesador.pdf

.

1.2 RISC-V

RISC-V is a free, open source hardware ISA licensed under BSD (Berkeley Software Distribution), based on the RISC (Reduced Instruction Set Computer) type design. The creation of the RISC-V architecture initially arose to support research in hardware architectures and to facilitate their learning. Being a free architecture, it allows us to design and manufacture chips using said architecture for various applications, free of charge, avoiding the payment of licenses [2]. This makes RISC-V an interesting

architecture, both at an educational and commercial level, compared to the aforementioned architectures.

RISC-V arose from the realization of several academic computer design projects, mainly oriented to research and teaching. The project began in 2010 at the University of Berkeley (California) with the aim of turning it into an open architecture standard for various applications. Professor Krste Asanović found many uses for an open source computer system. In 2010, he developed and published a "short three-month project over the summer" with the aim of helping academic and industrial users. David Patterson, his great collaborator, is considered one of the pioneers in RISC. In 1980, he led four generations of RISC projects (hence the name RISC-V); author of five books, two of the books on computer architecture and were written with Professor John Hennessy. Hennessy became a fellow at Stanford University in 1977. In 1984, he founded MIPS Computer Systems Inc to commercialize the RISC processors he had been researching on. Andrew Waterman co-designed the ISA RISC-V, is a major contributor to the RISC-V-based open source Rocket chip generator and the first RISC-V microprocessor. In addition, he is the co-founder of the SiFive company. SiFive was founded by the creators of the RISC-V architecture to provide custom chips based on RISC-V.

RISC-V is structured around its Foundation: “RISC-V Foundation”. Headquartered in Switzerland, it currently has more than 300 members (companies, entities, organizations ...), including firms such as Google, Qualcomm, NVIDIA, Samsung, Western Digital, IBM or Micron; but also others from outside the United States, such as European chipmaker NXP Semiconductors, e-commerce giant Alibaba Group, and Huawei Technologies or universities such as the Complutense University of Madrid.

One of the novelties that RISC-V provides is its modular and extensible design. The core of the ISA is the RV32I. The RV32I never changes and provides stability for compiler developers, operating systems, and assembly language programmers. Its modularity is produced thanks to standard optional extensions that the hardware can incorporate according to the need of the application to be developed. Additionally, its modularity enables very small and energy-efficient implementations of RISC-V.

Although it is not the first open architecture ISA, it is significant because it is designed to be useful and efficient on a wide range of devices, from the smallest to the fastest. RISC-V is a recent and open ISA, which takes advantage of the good ideas of the previous architectures, avoiding their errors.

In RISC-V all operations are performed between registers, except load / store, which do use memory accesses. RV32I has 32 general purpose registers that contain the status of the processor, the data that is being operated at all times, and the maintenance information. The register bank is able to minimize the need to access external memory for many basic CPU tasks, reducing power usage and increasing speed.

The fundamental objective of RISC-V is to become a universal ISA by being an open ISA, focused on maintaining the stability of RISC-V carefully and slowly, mainly based on technical reasons.

Next, in Table 1-2, the main technical characteristics of RISC-V [4] are shown:

Designer	University of California, Berkeley
Management	RISC-V Foundation.
License	BSD (open and free).
Word size	32-bit, 64-bit, 128-bit (with SIMD or vector extensions).
Type	RISC, load/store.
Core	RV32I, RV64I, RV128I (does not change, has modular extensions).
Modular extensions	M (instructions for multiplying and dividing). A (atomic operations). F (single precision floating point arithmetic). D (double precision floating point arithmetic). Q (quadruple precision floating point arithmetic). L (decimal floating point arithmetic). C (compressed instructions). B (bit manipulation). P (for SIMD instructions). V (vector). E (embedded systems applications) and G (set M, A, F and D).
Types of registers	General purpose (16 or 32, register x0: always at value 0x0).
Coding	Variable.
Branching	Comparison and jumps.
Endianness	Little-endian.
ISA modular	Modules can be added from the fixed ISA.

Table 1-2. Technical characteristics of RISC-V.
Source: (June 13, 2019). Architecnologia. Obtained frome https://architecnologia.es/risc-v-introduccion-a-la-isa-parte-2

In Chapter 2, a more in-depth analysis of the RISC-V architecture will be carried out: the study of the instruction set (the fundamental core of the ISA RV32I), the instruction format, the register bank, the addressing modes, as well as assembly language.

1.3 Motivation

Since my childhood I have always shown special interest in electronic devices, tools, cables... Although the current health crisis has made it quite difficult for me, innovating and overcoming myself are some of my challenges. So, I have no doubt that the RISC-V instruction set designed with fast, low-power real-world implementations in mind favor further innovation due to open market competition by many new designers.

My innate curiosity for the unknown, together with my continuous academic training provided by the Degree in Computer Engineering, has aroused my interest in the world of Hardware. In addition, my interest in the Computer Architecture subject increased this interest due to the proposal of my tutors. For this reason, I am convinced that this Final Degree Project enables me to know and deepen it.

Finally, I firmly believe that we must take advantage of its free hardware architecture RISC-V, this allows anyone with knowledge to participate in its development, to improve the future without additional costs, especially in these difficult times that we face.

1.4 Objectives

The main objective of this Final Degree Project is to carry out a study of the RISC-V architecture and its hardware and software ecosystem, based on all the information and documentation obtained for this purpose. In addition, the analysis and study of the SparkFun Red-V Thing Plus board will be carried out which uses the RISC-V architecture.

This document is structured in six chapters described below:

Chapter 1 - "Introduction". It begins with a very brief history of the RISC and CISC architectures before turning to the RISC-V architecture; the motivation that led to the realization of this project, the objectives together with the structure of the document and the work plan.

Chapter 2 - "Study of the repertoire of instructions". The most essential aspects of the RISC-V architecture such as the instruction format, the addressing modes, the register bank, the CPU modes and the assembly language will be analyzed.

Chapter 3 - "Study of the Software ecosystem". Some of the simulators, toolchains, debuggers and operating systems that support the RISC-V architecture will be explained.

Chapter 4 - "Study of the Hardware ecosystem". Some of the cores, SoCs, and boards that implement this instruction set will be covered.

Chapter 5 - "Use cases". The SparkFun Red-V Thing Plus board will be analyzed, through a basic tutorial on how to make a C program and assembler, with the board connected to the computer, using two different IDE: PlatformIO in Visual Studio Code and Freedom Studio from SiFive.

Chapter 6 - "Conclusions". Both the objective of the work and the technical conclusions derived from it will be analyzed, in addition to exposing the difficulties encountered and proposing ideas for possible future jobs.

Finally, the appendices will develop in greater depth some topics that, although they are not considered essential for the follow-up of this project, are considered of interest to the reader and whose objective is to expand the contents of the main document.

Definitely, the main objective of this Final Degree Project is to make a guide based on a compilation of information obtained through the study and research of different guides, manuals, documents, forums ... and so, to discover the benefits that can be provide a new architecture such as RISC-V.

1.5 Work plan

From the first moment, contact with the tutors of this FDP has been made through e-mails, face-to-face meetings or video calls.

In the initial phase, together with the tutors, the research project, the working method and the basic documentation to be taken into account to carry out said project were defined.

Next, the project was planned by searching, selecting and organizing the information. It was necessary to define the objectives, specify the tasks to be carried out and capture the information obtained in the chapters defined in section 1.4.

As knowledge of the investigated topic was acquired, the appropriate notes were taken to gradually write the report (reviewable at all times for pertinent modifications).

Subsequently, the study and analysis of the Sipeed Longan Nano and Arty A7 boards was carried out. Despite the time invested in them, no results were achieved due to difficulties caused by:

a) The confinement caused by the health emergency of COVID-19 had a strong impact on not being able to have the face-to-face means or the necessary materials from my home where I had to pass it.
b) The repeated attempts to operate the boards did not have the desired success, as would have been expected:

    - Partitions were made on the computer by installing Ubuntu due to lack of space on the hard disk, even using other computers to check its operation.
    - Tests were carried out with a voltmeter to measure the current in the different points of the board to check if they were defective or not.
    - Shortage of documentation in Spanish, the majority in Chinese, as well as the shortage of responses in the forums for this purpose.

Finally, the project was prepared by means of the writing of this report and its subsequent oral presentation.

Capítulo 2 - Estudio del repertorio de instrucciones

La información de ese capítulo es un resumen obtenido principalmente de *Patterson, D. y & Waterman, A. (2018). Guía práctica de RISC-V: El Atlas de una Arquitectura Abierta* y *Waterman, A. &* Asanović*, K. (December 13, 2019). The RISC-V Instruction Set Manual. Volume I: Unprivileged ISA.*

La estructura de un computador es la implementación mediante módulos y redes de conexión de las especificaciones del sistema dadas en la Arquitectura [5]. Dicha arquitectura se define mediante el repertorio de instrucciones, el cual, se puede ver como la frontera en la que el diseñador y el programador ven la misma máquina.

Desde el punto de vista del diseñador, el repertorio de instrucciones da las especificaciones funcionales de la Unidad Central de Proceso (CPU). El objetivo de los diseñadores de computadores es buscar un repertorio que facilite la construcción del hardware y del compilador, al tiempo que se maximiza el rendimiento y se minimiza el coste. Por ello, dicho repertorio debe ser completo, es decir, debe permitir calcular en tiempo finito cualquier tarea computable.

Por tanto, el repertorio de instrucciones se puede definir como el conjunto de todas las órdenes que puede ejecutar y entender la unidad de control de un computador en concreto. Incluye aspectos como el repertorio de operaciones, los tipos de datos, el formato de instrucciones, los modos de direccionamiento, la cantidad de registros, etc. Cada uno de estos aspectos será tratado en los siguientes apartados del presente Capítulo.

El enfoque convencional en arquitectura de computadores es desarrollar ISAs incrementales; de esta manera, los nuevos procesadores implementan tanto las nuevas extensiones, como todas las instrucciones de ISAs anteriores. Como se menciona anteriormente en el capítulo 1, una de las novedades que aporta RISC-V es su diseño **modular** y **extensible** [6]. El núcleo fundamental del ISA es el RV32I y permite la ejecución de programar a partir de un código de alto nivel. El RV32I nunca cambia y proporciona estabilidad para desarrolladores de compiladores, sistemas operativos y programadores de lenguaje ensamblador. Su diseño modular se produce gracias a extensiones opcionales estándar que el hardware puede incorporar según la necesidad de la aplicación que se vaya a desarrollar, lo que posibilita implementaciones muy pequeñas y de bajo consumo energético de RISC-V.

El ISA posee la mínima cantidad de instrucciones para facilitar la implementación física del hardware y la creación del software. Dicha arquitectura tiene como principal tipo de dato los números enteros, aunque soporta el uso de punto flotante con una extensión al conjunto de instrucciones.

RISC-V ha diseñado simultáneamente repertorios de instrucciones para tres tamaños de operandos. Esto significa que el número de bits de los registros del banco de registros es 32, 64 o 128, respectivamente. Las instrucciones se agrupan en módulos y, a la hora de diseñar un procesador RISC-V se pueden elegir los módulos que se implementarán, de forma que un procesador no necesita implementar todas las instrucciones.

Al compilador de RISC-V se le indica mediante las *flags* qué extensiones existen en hardware. La convención es concatenar las letras de extensión que son soportadas por dicho hardware. Para especificar las combinaciones de funcionalidad que pueden implementarse, se define una nomenclatura en el siguiente orden: la base del conjunto de instrucciones, el ancho de bits del banco de registros y la variante [RVXXY]; por ejemplo, RV32I o RV64I. Por último, las extensiones implementadas [RVXXYZ]:

- RV: base del conjunto de instrucciones (abreviación de RISC-V).
- XX: número de bits del banco de registros; pueden ser 32, 64 o 128 bits.
- Y: variante. El conjunto base es de operaciones con enteros (I).
- Z: extensiones que se pueden incluir. La Tabla 2-1 resume los nombres de extensión estandarizados y define el orden canónico de las extensiones que deben aparecer, siempre de arriba hacia abajo.

La **extensión estándar** está diseñada para no entrar en conflicto con ninguna otra estándar. Las primeras cuatro extensiones estándar para las bases enteras son: "M" para la multiplicación y división, "A" para instrucciones atómicas de memoria, "F" para instrucciones de punto flotante de precisión simple y "D" para instrucciones de coma flotante de doble precisión; por ejemplo:

> *RV32IMAFD, además de estar compuesto por las instrucciones base obligatorias RV32I, añade la multiplicación y división RV32M, las instrucciones atómicas RV32A, junto con punto flotante precisión simple RV32F y extensiones de punto flotante de precisión doble RV32D.*

- La letra “G” sustituye a IMAFD y queda definida para representar la base y las extensiones de "IMAFDZicsr_Zifencei" [7], o sea, el ISA estándar de uso general.

Algunas extensiones estándar de ISA dependen de la presencia de otras; por ejemplo, "D" depende de "F" y "F" depende de “Zicsr”. Estas dependencias pueden estar implícitas en el nombre ISA: por ejemplo, RV32IF es equivalente a RV32IFZicsr, y RV32ID es equivalente a RV32IFD y RV32IFDZicsr (véase Tabla 2-1).

Existen otras extensiones estándar de RISC-V ISA con letras reservadas [6]:

- La extensión “Q” agrega instrucciones de punto flotante binario de precisión cuádruple de 128 bits. Los registros de punto flotante almacenan un valor de

punto flotante de precisión simple, doble o cuádruple. La extensión de punto flotante binario de precisión cuádruple requiere RV64IFD.

- La extensión “C” agrega instrucciones en la extensión RVC para 16 bits: muchas instrucciones RV32C solo acceden a diez registros (a0–a5, s0–s1, sp y ra).

Además, la Fundación RISC-V espera desarrollar otras extensiones “opcionales” (Tabla 2-1), las cuales, a la publicación de *The RISC-V Instruction Set Manual. Volume I Unprivileged ISA* con fecha 13 de diciembre de 2019, todavía no habían sido ratificadas:

- La extensión “L” agrega instrucciones de punto flotante decimal. El problema con números binarios es que no pueden representar algunas fracciones decimales comunes.
- La extensión “B” ofrece manipulación de bits.
- La extensión “J” para Lenguajes Traducidos Dinámicamente, incluyendo Java y Javascript. (La letra J es para abreviar compilador *Just-In-Time*).
- La extensión “P” para instrucciones Packed-SIMD; este diseño reutiliza recursos de un datapath (ruta de datos) de ancho existente.
- La extensión “V”, que agrega instrucciones vectoriales.
- La extensión “E” para sistemas empotrados con 16 registros menos para reducir el costo de núcleos de gama baja.

Otras extensiones estándar a nivel de supervisor, hipervisor y máquina [7] son las mostradas en la Tabla 2-1. Debemos de tener en cuenta que si se enumeran varias extensiones, independientemente del nivel al que pertenezcan, deben aparecer después de las extensiones estándar con menos privilegios y ordenarse alfabéticamente.

- Extensiones del conjunto de instrucciones a nivel de supervisor. Se nombran usando "S" como prefijo.
- Extensiones de conjunto de instrucciones a nivel de hipervisor. Comienzan con la letra "H".
- Extensiones de conjunto de instrucciones a nivel de máquina. Tienen el prefijo de tres letras “Zxm”.

Por último, debido al crecimiento en el número de extensiones, la Fundación RISC-V ha ideado otras nomenclaturas que aparecen en el Apéndice A.

Nombre	Dependencia	Descripción	Versión	Estado
Base ISA				
I		Base para números enteros	2.1	*Ratified*
E		Sistemas Empotrados	1.9	*Draft*
Extensiones estándar				
M		Multiplicación y división de enteros	2.0	*Ratified*
A		Operaciones atómicas	2.1	*Ratified*
F	Zicsr	Punto flotante de precisión simple	2.2	*Ratified*
D	F	Punto flotante de precisión doble	2.2	*Ratified*
G	IMAFDZifencei	General		
Q	D	Punto flotante de precisión cuádruple	2.2	*Ratified*
L		Punto flotante decimal	0.0	*Draft*
C		Instrucciones comprimidas	2.0	*Ratified*
B		Manipulación de bits	0.0	*Draft*
J		Lenguajes traducidos dinámicamente	0.0	*Draft*
T		Memoria transaccional	0.0	*Draft*
P		Instrucciones SIMD empaquetados	0.2	*Draft*
V		Operaciones vectoriales	0.7	*Draft*
N		Interrupciones a nivel de usuario	1.1	*Draft*
Zicsr		Registro de control y estado (CSR)	2.0	*Ratified*
Zifencei		Instruction-Fetch Fence	2.0	*Ratified*
Zam	A	Atómicas desalineadas	0.1	*Draft*
Ztso		Total Store Ordering	0.1	*Frozen*
Extensiones estándar a nivel Supervisor				
Sdef		A nivel Supervisor “def”	1.11	*Ratified*
Extensiones estándar a nivel Hypervisor				
Hghi		A nivel Hypervisor “ghi”	1.11	*Ratified*
Extensiones estándar a nivel Máquina				
Zxmjkl		A nivel Maquina “jkl”	1.11	*Ratified*
Extensiones no estándar				
Xmno		Extensión no estándar “mno”	2.0	*Ratified*

Tabla 2-1. Extensiones estándar ISA
Fuente: Elaboración propia a partir de Waterman, A. & Krste Asanovi´c (December, 2019). The RISC-V Instruction Set Manual. Volume I: Unprivileged ISA.

Observación: Los módulos de ISA marcados como Ratified han sido ratificados en este momento. No se espera que los módulos marcados como Frozen cambien significativamente antes de someterse a la ratificación. Se espera que los módulos marcados como Draft cambien antes de la ratificación.

2.1 Formato de instrucciones

El formato de instrucción es el conjunto de especificaciones que indican cómo debe ser interpretado el patrón de *bits* de una instrucción máquina para lograr su ejecución dentro del computador [8]. Dicho formato nos indica cuál es el código de operación y cuáles los operandos que la instrucción especifica.

El formato de instrucción debe contener información como la operación que realiza la instrucción, la dirección de los operandos, la dirección resultado, la dirección de la siguiente instrucción y los tipos de representación de operandos [5].

Cuando se diseña un computador ha de elegirse el formato de instrucción. Una de las decisiones más importantes sobre el diseño del formato es la longitud (número de bits de la instrucción) porque afecta al número de campos, el tamaño de campos, el tamaño de memoria… Dicha decisión afecta y se ve afectada por el tamaño de la memoria (fija el tamaño de la palabra y el número de palabras de dicha memoria), la estructura y tamaño del bus, la complejidad y la velocidad de CPU.

Cuantos más códigos de operación, modos de direccionamiento, operandos y mayor rango de direcciones posea el repertorio de instrucciones, el programador escribirá programas con mayor facilidad. Dado que todos estos aspectos necesitan mayor longitud de instrucción, el diseñador del sistema tiene que buscar el equilibrio entre la riqueza del repertorio de instrucciones y la necesidad de tener espacio de memoria. Como norma general, la longitud de instrucción debe ser igual a la longitud de la transferencia de memoria o múltiplo de ella.

Existen seis formatos básicos [6] de instrucciones en el RV32I, ISA base, como se indica en la Figura 2-1, donde debemos tener en cuenta algunas consideraciones como:

- Todas las instrucciones son de 32 bits, simplificando la decodificación de instrucciones.
- En las instrucciones, salvo load y store, se dispone de tres registros para evitar que el compilador o programador de ensamblador use una instrucción adicional para guardar el operando destino.
- Los bits de los registros a ser leídos y escritos están en la misma posición para todas las instrucciones, así se puede empezar a acceder a dichos registros antes de la decodificación.
- Los campos inmediatos en estos formatos siempre son extendidos en signo, y el bit del signo siempre está en el bit más significativo de la instrucción. Esta

decisión implica que la extensión de signo del inmediato puede continuar antes de la decodificación.

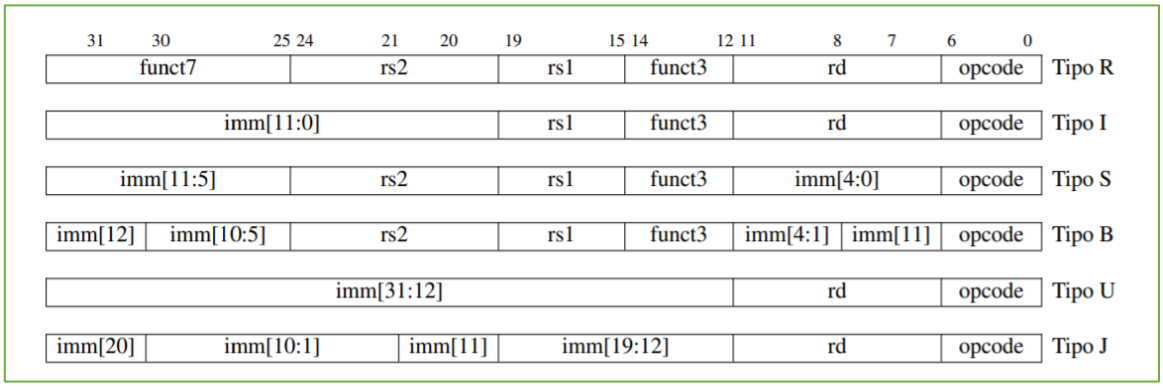

Figura 2-1. Formato de instrucciones.
Fuente. Patterson, D., & Waterman, A. (2018). Guía práctica de RISC-V: El Atlas de una Arquitectura Abierta. Strawberry Canyon LLC.

Los seis formatos básicos de instrucción RV32I son los siguientes:

- Tipo R: instrucciones para operaciones entre registros. Se leen los valores contenidos en dos registros de 32 bits y se escribe el resultado en el registro destino.
- Tipo I: instrucciones para inmediatos cortos y loads.
- Tipo S: instrucciones para stores (accesos a memoria).
- Tipo B: instrucciones de control de flujo (branches).
- Tipo U: instrucciones para inmediatos largos.
- Tipo J: instrucciones para saltos incondicionales.

El campo funct3 es el identificador de la operación a realizar y las instrucciones (dependiendo del tipo) reparten sus **32 bits de longitud** de esta forma:

- Los 7 primeros bits (0-6) al opcode, es decir, el que determina el tipo de operación implícita en la instrucción.
- El registro destino se marca con un código compuesto de 5 bits (7-11). No en todas está presente este campo.
- Se marca el primer registro fuente donde está el operando. Cuando está presente siempre estará en los siguientes 5 bits (15-19).
- El segundo registro fuente, si está presente, estará siempre en los 5 bits que van del 20 al 24.
- También pueden contener valores inmediatos, es decir, un valor constante.
- Los campos de función determinan la variante de la operación establecida en el campo de opcode.

RV32I es el repertorio de instrucciones más utilizado por la facilidad de diseño de hardware y la implementación del software. La Figura 2-2 muestra para 47 instrucciones,

la estructura de la instrucción, opcodes, tipo de formato y nombres. Dicha figura muestra el mapa de opcodes usando los formatos de la Figura 2-1.

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	R sltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and
0000 pred	succ	00000	000	00000	0001111	I fence
0000 0000	0000	00000	001	00000	0001111	I fence.i
000000000000		00000	000	00000	1110011	I ecall
000000000001		00000	000	00000	1110011	I ebreak
csr		rs1	001	rd	1110011	I csrrw
csr		rs1	010	rd	1110011	I csrrs
csr		rs1	011	rd	1110011	I csrrc
csr		zimm	101	rd	1110011	I csrrwi
csr		zimm	110	rd	1110011	I csrrsi
csr		zimm	111	rd	1110011	I csrrci

Figura 2-2. Mapa de opcodes de RV32I.
Fuente. Elaboración propia a partir de Patterson, D., & Waterman, A. (2018). Guía práctica de RISC-V: El Atlas de una Arquitectura Abierta. Strawberry Canyon LLC.

2.1.1 Computación Entera

Para la computación que involucre datos enteros se utilizan las instrucciones de tipo R. Las instrucciones aritméticas sencillas (add, sub), las instrucciones lógicas (and, xor, or), de comparación (slt, sltu) y las instrucciones de desplazamiento (sll, srl, sra) realizan la función deseada, leen dos valores de registros de 32 bits y escriben el resultado al registro destino. RV32I también posee versiones inmediatas de estas instrucciones (addi, slli, slti, sltiu, xori, srli, srai, ori, andi).

2.1.2 Loads y Stores

Para los loads y stores se utilizan las instrucciones de tipo S y tipo I. RV32I no sólo dispone de *loads y stores* de palabras de 32 bits (lw, sw), sino que también puede cargar bytes y halfwords, en versión *signed o unsigned* (lb, lbu, lh, lhu) y guardar *bytes y halfwords* (sb, sh). *Bytes y halfwords* con signo tienen 32 bits y son escritos en el registro destino. Aunque el dato original sea más corto de 32 bits, esta extensión de datos permite que las operaciones aritméticas posteriores operen correctamente. *Bytes y halfwords* sin signo, se extienden con cero a 32 bits.

2.1.3 Saltos Condicionales

Los saltos condicionales se gestionan mediante instrucciones de tipo B. En RV32I se puede comparar dos registros y saltar si el resultado es igual (beq), distinto (bne), mayor o igual (bge), o menor (blt). Además, también están disponibles las versiones sin signo (bgeu, bltu).

2.1.4 Saltos Incondicionales

Los saltos incondicionales se gestionan mediante instrucciones de tipo J y tipo I. La instrucción *jump and link* (jal) para llamadas a funciones almacena la dirección de la siguiente instrucción PC+4 en el registro destino, normalmente el registro *ra*. Para los saltos incondicionales utiliza el registro cero (x0) en lugar de *ra* ya que no cambia.

Además, también exite la instrucción *jump and link* con registro (jalr); esta puede hacer una llamada a función a una dirección de memoria calculada dinámicamente o retornar de la función usando *ra* como registro origen, y el registro cero como destino. La instrucción *jalr* resulta útil para operaciones de *switch o case*, que calculan la dirección a saltar.

2.1.5 Miscelánea

Para este tipo se utilizan las instrucciones de tipo I. Las instrucciones de los registros de control y estado (*control status register):* csrrc, csrrs, csrrw, csrrci, csrrsi, csrrwi proporcionan el acceso a registros que ayudan a medir el rendimiento de un programa. Estos registros actúan como contadores de 64 bits: miden el tiempo, ciclos ejecutados y número de instrucciones retiradas.

Los CSRs de medición [6] son los siguientes:

- De Tiempo de Máquina mtime: contador de tiempo real de 64 bits.
- De Comparación de Tiempo de Máquina mtimecmp: causa una interrupción cuando mtime iguala o excede su valor.
- De habilitación de contadores de máquina y supervisor de 32 bits (mcounteren y scounteren) controlan la disponibilidad de los CSRs monitores de rendimiento de hardware al siguiente nivel menos privilegiado.
- Los 32 CSRs monitores de rendimiento de hardware (mcycle, minstret, mhpmcounter3, ..., mhpmcounter31) cuentan ciclos de reloj, instrucciones retiradas, y hasta 29 eventos seleccionados por el programador usando los CSRs mhpmevent3, ..., mhpmevent31.

Por otro lado, la instrucción *ecall* hace peticiones al entorno de ejecución, como llamadas al sistema. Los depuradores utilizan la instrucción *ebreak* para transferir el control al entorno de depuración. La instrucción *fence* coordina accesos a dispositivos de I/O. La instrucción *fence.i* sincroniza el flujo de instrucciones y datos.

A continuación, en la Figura 2-3 nos muestra una representación gráfica del conjunto de instrucciones base RV32I mostrado anteriormente en el Mapa de Opcodes (Figura 2-2). Las letras subrayadas son concatenadas de izquierda a derecha para formar instrucciones RV32I. La notación de llaves { } implica que cada ítem vertical en el conjunto es una variante diferente de la instrucción, utilizando letras subrayadas o guión bajo_, esto quiere decir que no hay que agregar letras en esta variante.

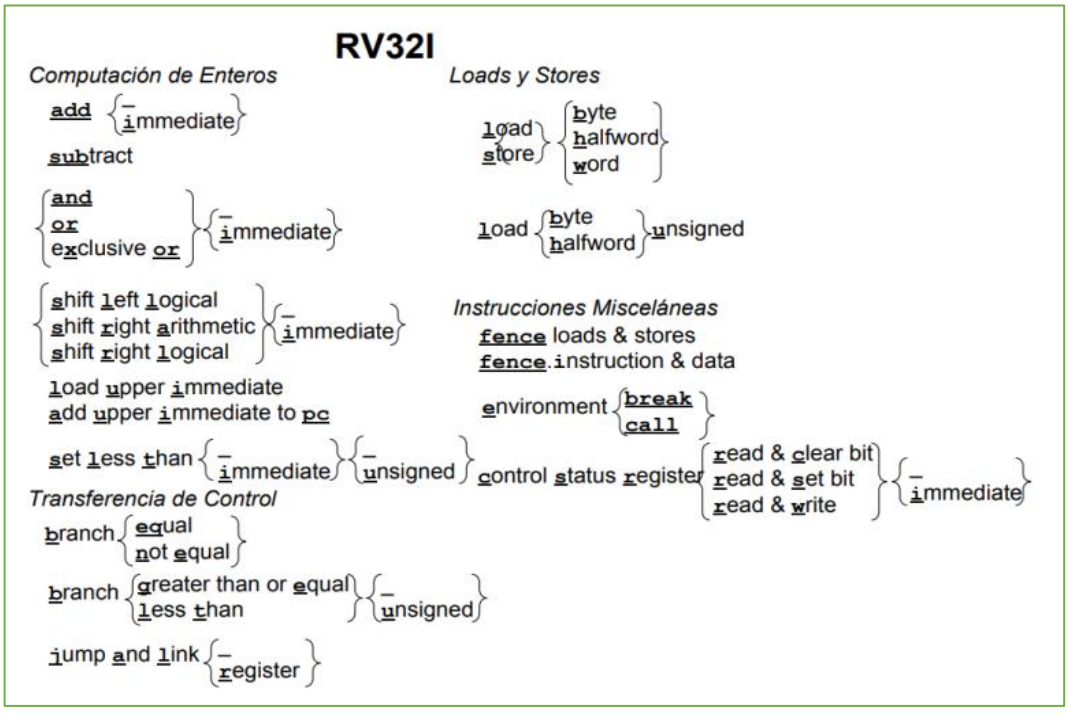

Figura 2-3. Diagrama de las instrucciones RV32I.
Fuente. Patterson, D., & Waterman, A. (2018). Guía práctica de RISC-V: El Atlas de una Arquitectura Abierta. Strawberry Canyon LLC.

Por poner un ejemplo, si considerásemos una notación de la Figura 2-3, veríamos como dicha notación (Figura 2-4) representa estas seis instrucciones RV32I: *and, or, xor, andi, ori, xori.*

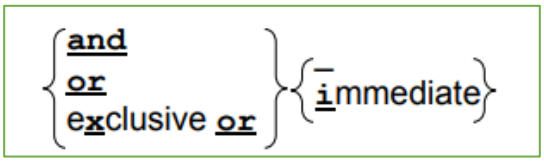

Figura 2-4. Ejemplo sobre instrucciones RV32I.
Fuente: Elaboración propia a partir de Patterson, D., & Waterman, A. (2018). Guía práctica de RISC-V: El Atlas de una Arquitectura Abierta. Strawberry Canyon LLC.

Para finalizar con este apartado, a continuación, en la Tabla 2-2, se muestran algunos ejemplos de instrucciones junto con su significado [9]:

Instrucción	Significado
$\boldsymbol{sub\ x2, x3, x4}$	$Reg[x2] \leftarrow Reg[x3] - Reg[x4]$
$\boldsymbol{addi\ x2, x3, 6}$	$Reg[x2] \leftarrow Reg[x3] + 6$
$\boldsymbol{sll\ x2, x3, 3}$	$Reg[x2] \leftarrow Reg[x3] \ll 3$
$\boldsymbol{ld\ x1, 80\ (x2)}$	$Reg[x1] \leftarrow Mem[80 + Reg[x2]]$
$\boldsymbol{sb\ x3, 41(x4)}$	$Mem[41 + Reg[x4]] \leftarrow_8 Reg[x3]$
$\boldsymbol{lui\ x2, 42}$	$Reg[x2] \leftarrow 0^{32}\#\#42\#\#0^{12}$
$\boldsymbol{beq\ x3, x4, offset}$	$if(Reg[x3] == Reg[x4])PC \leftarrow PC + (offset \ll 1)$
$\boldsymbol{jalr\ x1, x2, offset}$	$Regs[x1] \leftarrow PC + 4; PC \leftarrow Reg[x2] + offset$
$\boldsymbol{ecall}$	$Atomic\ jump\ to\ location\ 0x80000180$
$\boldsymbol{csrr\ x5, mstatus}$	$x5 \leftarrow mstatus$

Tabla 2-2. Ejemplos de instrucciones.
Fuente: Elaboración propia a partir de Hennessy, J. & Patterson, D. (2019). Computer Architecture: A Quantitative Approach. (Sixth Edition). Morgan Kaufmann.

2.2 Modos de direccionamiento

Los modos de direccionamiento son los diferentes procedimientos que determinan la ubicación de un operando o una instrucción [10]. Las arquitecturas de computadores varían mucho en cuanto al número de modos de direccionamiento que ofrecen desde el hardware. Se ha comprobado que cuando los modos de direccionamiento son simples, el diseño de CPUs segmentadas es mucho más fácil.

Cuando existen pocos modos, van codificados directamente dentro de la propia instrucción (IBM/390, y en la mayoría de los RISC); pero cuando hay demasiados ocurre al contrario, ya que a menudo tiene un campo específico en la propia instrucción para especificar dicho modo de direccionamiento.

La mayoría de las máquinas RISC disponen de, apenas, cinco modos de direccionamiento simple [6]. RV32I suprime los modos de direccionamiento de ARM32 y x86-32. Aunque los modos de direccionamiento de ARM-32 no están disponibles para todos los tipos de datos, los modos de RV32I no discriminan a ningún tipo de dato. Sin embargo, RISC-V puede imitar algunos modos de direccionamiento del x86-32. Por ejemplo, dejando el valor inmediato en cero, es análogo al modo registro-indirecto.

En RISC-V nos encontramos principalmente con cuatro modos de direccionamiento:

- Indirecto. El único modo de direccionamiento para loads y stores consiste en sumar un valor inmediato de 12 bits a un registro, en x86-32 se denomina “modo de direccionamiento con desplazamiento” [6].
- Direccionamiento relativo al PC. Se usa en las instrucciones de saltos. Dado que las instrucciones de RISC-V deben ser múltiplos de dos bytes el modo de direccionamiento de saltos multiplica el valor inmediato de 12 bits por 2, le extiende el signo y lo suma al PC. En la instrucción jal se usa este mismo modo, multiplicando la dirección de 20 bits por 2, se extiende el signo y se suma el resultado al PC para obtener la dirección a saltar.
- Directo a registro. El operando se encuentra en un registro.
- Inmediato. El operando está especificado directamente en la instrucción.

A diferencia del x86-32, en RISC-V no existen instrucciones de stack específicas. Utilizando un registro como stack pointer, el modo de direccionamiento estándar obtiene la mayoría de los beneficios de las instrucciones push y pop.

2.3 Banco de Registros

En RISC-V todas las operaciones se realizan entre registros, salvo load/store que sí utilizan accesos a memoria. RV32 tiene 32 registros de propósito general (véase Tabla 2-3) de 32 bits nombrados del x0 al x31. La notación es muy sencilla *"xN"* siendo *N* el número de registro. Por ejemplo, el registro *x1* es el registro número 1.

El registro x0 siempre contiene un valor constante, 0x0. No se le puede asignar otro valor, se utiliza principalmente para inicializar otros registros a 0. En RV64 se utilizan 64 bits y en RV128, 128 bits, pero siempre son 32 registros. Además, las instrucciones del *control status register*, ya mencionadas anteriormente, proveen acceso fácil a registros que ayudan a medir el rendimiento de un programa.

El registro PC (program counter), contiene la dirección de la siguiente instrucción a ejecutar. En la Tabla 2-3 [11] se pueden ver los diferentes registros, algunos de ellos estarán resaltados con **letra negrita**, son los llamados *caller*, es decir, hay que preservarlos cuando se realiza una llamada a otra función.

Registro	Nombre	Significado
x0	zero	Valor constante a 0x0.
x1	ra	Return Address: posee la dirección de retorno de la función actual.
x2	sp	Stack Pointer: puntero a la cima de la pila.
x3	gp	Global Pointer: puntero a la sección global de Data.
x4	tp	Thread Pointer: puntero a la sección de punteros.
x5	t0	Temporaries: registros de propósito general que no se conservan en llamadas a funciones.
x6	t1	
x7	t2	
x8	s0 / fp	Saved Register / Frame Pointer: registro cuyo valor debe ser conservado entre llamadas. El FP contiene la dirección justo antes de la llamada a una función. Sirve para regresar el stack a su posición original al regresar de una función.
x9	s1	Saved Register: registro de propósito general, se debe conservar su valor entre llamadas.
x10	a0	Function Arguments / Return Values: registros utilizados para enviar argumentos en las llamadas a funciones. También son usados como valor de retorno de las funciones.
x11	a1	
x12	a2	Function Arguments: registros utilizados para enviar argumentos en las llamadas a funciones.
x13	a3	
x14	a4	
x15	a5	
x16	a6	
x17	a7	
x18	s2	Saved Registers: registros seguros de propósito general, se debe conservar su valor entre llamadas.
x19	s3	
x20	s4	
x21	s5	
x22	s6	
x23	s7	
x24	s8	
x25	s9	
x26	s10	
x27	s11	
x28	t3	Temporaries: registros de propósito general que no se conservan en llamadas a funciones.
x29	t4	
x30	t5	
x31	t6	

Tabla 2-3. Registros de RV32I. Convención de registros a preservar a través de llamadas a funciones. Fuente: Elaboración propia a partir de Estructura de máquinas - Introducción a RISC-V. (s.d.). Obtenido de https://cc-3.github.io/notes/02_Intro-RISCV/

2.4 Modos de la CPU

La Unidad Central de Procesamiento (Central Processing Unit) es el hardware de un ordenador u otros dispositivos programables que interpreta las instrucciones de un programa informático mediante operaciones aritméticas, lógicas y externas (de entrada/salida) [12]. Aunque la forma, el diseño y la implementación de las CPU ha cambiado totalmente con el paso del tiempo, su función sigue siendo la misma.

Todas las instrucciones mostradas hasta ahora están disponibles en el modo usuario, donde normalmente se ejecuta el código de aplicaciones.

RISC-V posee otros dos nuevos modos más privilegiados que el modo Usuario: modo Máquina que ejecuta el código más fiable y el modo Supervisor que provee soporte para sistemas operativos como Linux, FreeBSD y Windows (Tabla 2-4).

Estos modos más privilegiados tienen acceso a las características de los modos menos privilegiados y agregan funcionalidad en los mismos: gestionar excepciones, manejar interrupciones y controlar I/O [6]. Los procesadores pasan la mayoría del tiempo de ejecución en su modo menos privilegiado; las interrupciones y excepciones transfieren el control a modos más privilegiados. Los sistemas operativos utilizan los nuevos modos para responder a eventos externos, proteger tareas y abstraer características del hardware.

Codificación	Nombre	Abreviación
00	Usuario	U
01	Supervisor	S
11	Máquina	M

Tabla 2-4. Codificación de los niveles de privilegio en RISC-V.
Fuente: Elaboración propia a partir de Patterson, D., & Waterman, A. (2018).
Guía práctica de RISC-V: El Atlas de una Arquitectura Abierta. Strawberry Canyon

La Figura 2-5 muestra las instrucciones privilegiadas que existen en RISC-V y la Figura 2-6 sus opcodes.

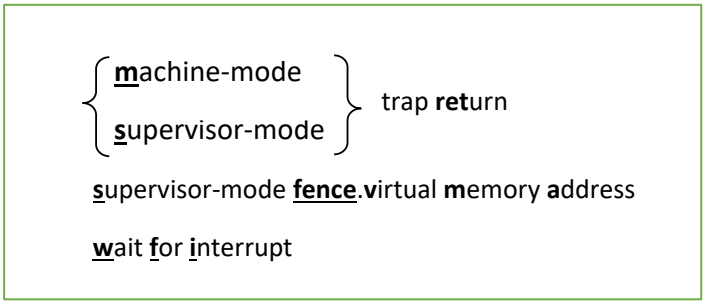

Figura 2-5. Diagrama de las instrucciones privilegiadas RISC-V. Fuente: Elaboración propia a partir de Patterson, D., & Waterman, A. (2018). Guía práctica de RISC-V: El Atlas de una Arquitectura Abierta. Strawberry Canyon.

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
0001000		00010	00000	000	00000	1110011	R sret
0011000		00010	00000	000	00000	1110011	R mret
0001000		00101	00000	000	00000	1110011	R wfi
0001001		rs2	rs1	000	00000	1110011	R sfence.vma

Figura 2-6. Estructura de instrucciones privilegiadas RISC-V, opcodes, tipo de formato y nombre. Fuente: Elaboración propia a partir de Patterson, D., & Waterman, A. (2018). Guía práctica de RISC-V: El Atlas de una Arquitectura Abierta. Strawberry Canyon.

2.4.1 Modo Máquina

Es el modo más privilegiado, en el que un *Hart* (Hardware thread: Hilo de ejecución en hardware) tiene acceso completo a la memoria, I/O, y funcionalidades del sistema [6]. La característica más importante del modo Máquina es la capacidad de interceptar y manejar las excepciones mostradas en la Figura 2-7 donde el bit más significativo de *mcause* es puesto en 1 para interrupciones o en 0 para excepciones síncronas.

El modo Máquina soporta aplicaciones empotradas básicas a un bajo coste. Posee los privilegios más altos y es el único nivel de privilegio obligatorio para una plataforma hardware basada en RISC-V. El código que se ejecuta en modo máquina (modo M) suele ser de confianza y se puede utilizar para gestionar la ejecución segura en entornos RISC-V.

En RISC-V existen ocho registros de control y estado (CSRs) para el manejo de excepciones en modo máquina que aparecen detalladamente en el [Apéndice B].

Interrupción / Excepción mcause[XLEN-1]	Código de Excepción mcause[XLEN-2:0]	Descripción
1	1	Interrupción de software de Supervisor
1	3	Interrupción de software de Máquina
1	5	Interrupción de temporizador de Supervisor
1	7	Interrupción de temporizador de Máquina
1	9	Interrupción externa de Supervisor
1	11	Interrupción externa de Máquina
0	0	Dirección de instrucción desalineada
0	1	Fallo de acceso en instrucción
0	2	Instrucción ilegal
0	3	Breakpoint
0	4	Dirección de Load desalineada
0	5	Fallo de acceso en Load
0	6	Dirección de Store desalineada
0	7	Fallo de acceso en Store
0	8	Llamada al entorno desde modo U
0	9	Llamada al entorno desde modo S
0	11	Llamada al entorno desde modo M
0	12	Fallo de página en instrucción
0	13	Fallo de página en Load
0	15	Fallo de página en Store

Figura 2-7. Causas de excepciones e interrupciones en RISC-V.
Fuente: Patterson, D., & Waterman, A. (2018). Guía práctica de RISC-V: El Atlas de una Arquitectura Abierta. Strawberry Canyon.

2.4.2 Modo Usuario

RISC-V provee mecanismos para proteger al sistema del código no confiable [6]. Para dicho código no confiable se debe prohibir ejecutar instrucciones privilegiadas porque permitirían al programa tomar el control del sistema. Estas restricciones se logran con el modo Usuario (véase Tabla 2-4), que niega el acceso generando una excepción de instrucción ilegal cuando se intenta usar una instrucción o CSR de modo M.

Además, también es necesario restringir el acceso del código no confiable a su propia memoria. Para ello, el procesador tiene implementada la llamada *Protección Física de Memoria* (PMP) *(Physical Memory Protection)*, que permite al modo M especificar a qué direcciones de memoria puede acceder el modo U. PMP posee varios registros de dirección que indican si conceden o niegan permisos de lectura, escritura y ejecución. El modo adicional Usuario y la Protección de Memoria Física juntos habilitan la multitarea en sistemas empotrados más sofisticados.

Para obtener una información detallada sobre dichos registros de dirección y configuración PMP, véase el [Apéndice B].

2.4.3 Modo Supervisor

El modo supervisor está diseñado para soportar sistemas operativos modernos similares a Unix, tales como Linux, FreeBSD y Windows [6]. El modo S es más privilegiado que el modo U, pero menos privilegiado que el modo M. Al igual que en el modo U, el software del modo S no puede usar CSRs ni instrucciones del modo M, y está sujeto a restricciones de PMP.

Además, RISC-V dispone de un mecanismo por el cual las interrupciones y excepciones síncronas pueden ser delegadas al modo supervisor selectivamente, evitando software

de modo máquina por completo, permitiendo reducir el tiempo que se invierte en manejar estas excepciones. [Apéndice B].

2.5 Lenguaje ensamblador

El lenguaje ensamblador es la representación simbólica de la codificación binaria de un computador, el lenguaje máquina [13]. Una herramienta llamada ensamblador traduce lenguaje ensamblador a instrucciones binarias. Los lenguajes ensambladores ofrecen una representación más próxima al programador que los ceros y los unos del computador.

El lenguaje ensamblador incluye las instrucciones de RISC-V y algunas instrucciones útiles externas [6]. La tarea del ensamblador, tal y como se muestra en la Figura 2-8, es producir código objeto a partir de instrucciones que el procesador pueda ejecutar y extenderlas para incluir operaciones útiles para el programador de lenguaje ensamblador o el desarrollador de compiladores. Esta categoría basada en variantes de instrucciones normales es lo que llamamos *pseudo-instrucciones*, las cuales se muestran en las Figuras 2-9 y 2-10.

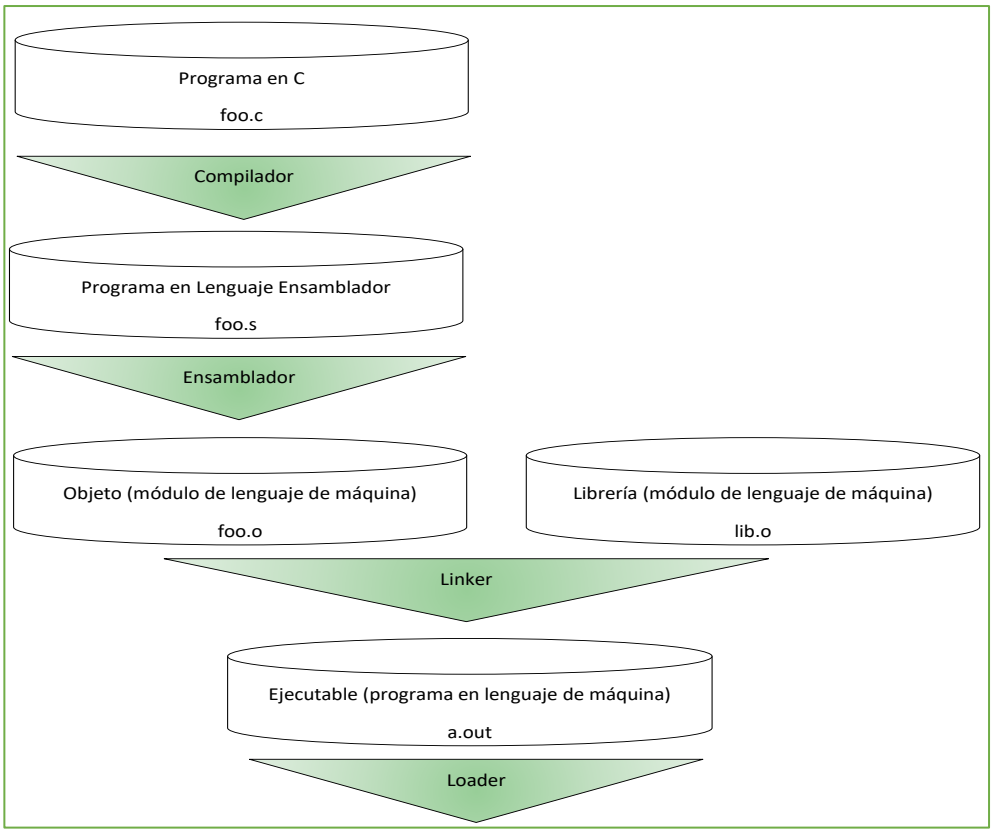

Figura 2-8. Pasos para convertir desde código fuente hasta un programa en ejecución
Fuente: Elaboración propia a partir de Patterson, D., & Waterman, A. (2018). Guía práctica de RISC-V: El Atlas de una Arquitectura Abierta. Strawberry Canyon LLC.

Pseudoinstrucción	Instrucción/Instrucciones Base	Significado
`nop`	`addi x0, x0, 0`	No operation
`neg rd, rs`	`sub rd, x0, rs`	Complemento a 2
`negw rd, rs`	`subw rd, x0, rs`	Complemento a 2 (word)
`snez rd, rs`	`sltu rd, x0, rs`	Poner en 1 si $\neq$ cero
`sltz rd, rs`	`slt rd, rs, x0`	Poner en 1 si $<$ cero
`sgtz rd, rs`	`slt rd, x0, rs`	Poner en 1 si $>$ cero
`beqz rs, offset`	`beq rs, x0, offset`	Branch si $=$ cero
`bnez rs, offset`	`bne rs, x0, offset`	Branch si $\neq$ cero
`blez rs, offset`	`bge x0, rs, offset`	Branch si $\leq$ cero
`bgez rs, offset`	`bge rs, x0, offset`	Branch si $\geq$ cero
`bltz rs, offset`	`blt rs, x0, offset`	Branch si $<$ cero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch si $>$ cero
`j offset`	`jal x0, offset`	Jump
`jr rs`	`jalr x0, rs, 0`	Jump a registro
`ret`	`jalr x0, x1, 0`	Retornar de subrutina
`tail offset`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	*Tail call* subrutina lejana
`rdinstret[h] rd`	`csrrs rd, instret[h], x0`	Leer el contador de instrucciones retiradas
`rdcycle[h] rd`	`csrrs rd, cycle[h], x0`	Leer el contador de ciclos
`rdtime[h] rd`	`csrrs rd, time[h], x0`	Leer *real-time clock*
`csrr rd, csr`	`csrrs rd, csr, x0`	Leer CSR
`csrw csr, rs`	`csrrw x0, csr, rs`	Escribir CSR
`csrs csr, rs`	`csrrs x0, csr, rs`	Poner bits en 1 en CSR
`csrc csr, rs`	`csrrc x0, csr, rs`	Poner bits en 0 en CSR
`csrwi csr, imm`	`csrrwi x0, csr, imm`	Escribir CSR, inmediato
`csrsi csr, imm`	`csrrsi x0, csr, imm`	Poner bits en 1 en CSR, inmediato
`csrci csr, imm`	`csrrci x0, csr, imm`	Poner bits en 0 en CSR, inmediato
`frcsr rd`	`csrrs rd, fcsr, x0`	Leer *FP control/status register*
`fscsr rs`	`csrrw x0, fcsr, rs`	Escribir *FP control/status register*
`frrm rd`	`csrrs rd, frm, x0`	Leer *FP rounding mode*
`fsrm rs`	`csrrw x0, frm, rs`	Escribir *FP rounding mode*
`frflags rd`	`csrrs rd, fflags, x0`	Leer *FP exception flags*
`fsflags rs`	`csrrw x0, fflags, rs`	Escribir *FP exception flags*

Figura 2-9. 32 pseudo-instrucciones de RISC-V que dependen de x0, el registro cero.
Fuente. Patterson, D., & Waterman, A. (2018). Guía práctica de RISC-V: El Atlas de una Arquitectura Abierta. Strawberry Canyon LLC.

Pseudoinstrucción	Instrucción/Instrucciones Base	Significado
`lla rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load de dirección local
`la rd, symbol`	*PIC*: `auipc rd, GOT[symbol][31:12]` `l{w|d} rd, rd, GOT[symbol][11:0]` *No-PIC*: Igual que `lla rd, symbol`	Load de dirección
`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]` `l{b|h|w|d} rd, symbol[11:0](rd)`	Load global
`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `s{b|h|w|d} rd, symbol[11:0](rt)`	Store global
`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fl{w|d} rd, symbol[11:0](rt)`	Load global de punto flotante
`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fs{w|d} rd, symbol[11:0](rt)`	Store global de punto flotante
`li rd, immediate`	*Muchas secuencias*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copiar registro
`not rd, rs`	`xori rd, rs, -1`	Complemento a uno
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Poner en 1 si = cero
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copiar registro de precisión simple
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Valor absoluto de precisión simple
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Negación de precisión simple
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copiar registro de precisión doble
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Valor absoluto de precisión doble
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Negación de precisión doble
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch si $>$
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch si $\leq$
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch si $>$, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch si $\leq$, unsigned
`jal offset`	`jal x1, offset`	Jump and link
`jalr rs`	`jalr x1, rs, 0`	Jump and link a registro
`call offset`	`auipc x1, offset[31:12]` `jalr x1, x1, offset[11:0]`	Llamar subrutina lejana
`fence`	`fence iorw, iorw`	*Fence* en toda la memoria e I/O
`fscsr rd, rs`	`csrrw rd, fcsr, rs`	Swap con *FP control/status register*
`fsrm rd, rs`	`csrrw rd, frm, rs`	Swap con *FP rounding mode*
`fsflags rd, rs`	`csrrw rd, fflags, rs`	Swap con *FP exception flags*

Figura 2-10. 28 pseudoinstrucciones de RISC-V que son independientes de x0, el registro cero.
Fuente. Patterson, D., & Waterman, A. (2018). Guía práctica de RISC-V: El Atlas de una Arquitectura Abierta. Strawberry Canyon LLC.

Teniendo en cuenta los pasos para convertir desde código fuente hasta un programa en ejecución de la Figura 2-8, la Figura 2-11 muestra el primer programa ejecutado "Hola Mundo" en C.

```
#include <stdio.h>
int main()
{
   printf("Hello, %s\n", "world");
   return 0;
}
```

Figura 2-11. Programa Hola Mundo en C (hello.c)
Fuente. Patterson, D., & Waterman, A. (2018). Guía práctica de RISC-V: El Atlas de una Arquitectura Abierta. Strawberry Canyon LLC.

El compilador produce el código en ensamblador de la Figura 2-12 usando la convención de llamadas de la Tabla 2-3 y las pseudoinstrucciones de las Figuras 2-9 y 2-10.

```
  .text                    # Directiva: ingresar a sección text
  .align 2                 # Directiva: alinear código a 2^2 bytes
  .globl main              # Directiva: declarar símbolo global main
main:                      # etiqueta para inicio de main
  addi sp,sp,-16           # reservar stack frame
  sw   ra,12(sp)           # guardar dirección de retorno
  lui  a0,%hi(string1)     # calcular dirección de
  addi a0,a0,%lo(string1)  #   string1
  lui  a1,%hi(string2)     # calcular dirección de
  addi a1,a1,%lo(string2)  #   string2
  call printf              # llamar función printf
  lw   ra,12(sp)           # restaurar dirección de retorno
  addi sp,sp,16            # liberar stack frame
  li   a0,0                # cargar valor de retorno 0
  ret                      # retornar
  .section .rodata         # Directiva: ingresar a sección read-only data
  .balign 4                # Directiva: alinear sección data a 4 bytes
string1:                   # etiqueta para el primer string
  .string "Hello, %s!\n"   # Directiva: null-terminated string
string2:                   # etiqueta para el segundo string
  .string "world"          # Directiva: null-terminated string
```

Figura 2-12. Programa Hola Mundo en lenguaje ensamblador de RISC-V (hello.s).
Fuente. Patterson, D., & Waterman, A. (2018). Guía práctica de RISC-V: El Atlas de una Arquitectura Abierta. Strawberry Canyon LLC.

Los comandos que comienzan con un punto son directivas del ensamblador. Estos son comandos para el ensamblador, no es un código para traducir. Le indican al ensamblador dónde poner código y datos, especifican constantes de texto y datos para uso en el programa, etc; por lo que, las directivas son órdenes que afectan al ensamblador. Dichas directivas permiten reservar memoria con un valor determinado, definir símbolos, procedimientos o subrutinas que hagan más legible el programa, marcar el inicio y el

final del programa, iniciar variables... Aparte de escribir el código del algoritmo mediante instrucciones de ensamblador, el programador en lenguaje ensamblador debe reservar espacio en memoria para las variables, y si tuvieran un valor inicial, escribir este valor en la dirección correspondiente.

Dado que en los posteriores capítulos se utilizará código ensamblador, a continuación, se enumerarán las directivas más comunes del ensamblador RISC-V [6]:

- Directivas para declarar secciones del ejecutable: definen la sección del programa.
    - .text: código del programa.
    - .data: variables globales con valor inicial.
    - .bss: variables globales sin valor inicial.
    - .section .foo: sección llamada .foo.
- Alineamiento de datos:
    - .align n: alinear el siguiente dato en un límite de 2^n bytes.
    - .balign n: alinear el siguiente dato en un límite de n bytes.
- Símbolos globales:
    - .global sym: exporta un símbolo para que pueda usarse desde otros archivos. El comienzo del programa se utiliza mediante la directiva .global _start.
- Reservas de dirección de memoria:
    - .string “str”: almacena el string str en memoria.
    - .byte b1,…, bn: almacena las n cantidades de 8 bits en bytes sucesivos de memoria.
    - .half w1,…,wn:almacena las n cantidades de 16 bits en halfwords sucesivos de memoria.
    - .word w1,…, wn: almacena las n cantidades de 32 bits en words sucesivos de memoria.
    - .dword w1,…,wn: almacena las n cantidades de 64 bits en doublewords sucesivos de memoria.
    - .float f1,…,fn: almacena los n números de punto flotante de precisión simple en words sucesivos de memoria.
    - .double d1,…,dn: almacena los n números de punto flotante de precisión doble en doublewords sucesivos de memoria.

- Directivas de opciones:
    - .option rvc: comprime algunas instrucciones.
    - .option norvc: no comprime instrucciones.
    - .option relax: permite relajación del linker para instrucciones.
    - .option norelax: no permite relajación del linker para instrucciones.
    - .option pic: las instrucciones subsiguientes son código independiente de posición.
    - .option nopic: las instrucciones subsiguientes son *position-dependent code.*
    - .optionpush: hacer push del estado de todos los .options a un stack, posteriormente se realiza un .option pop para restaurar sus valores.
    - .option pop: hacer pop del stack de todos los .options a su estado en el momento del último .optionpush.

Para finalizar con el presente capítulo, es preciso indicar que RISC-V requiere un hardware capaz de ejecutar las instrucciones del repertorio. La arquitectura de ISA abierto consigue reducir esa complejidad de los repertorios de instrucciones tan presentes en el mercado. Dicho repertorio permite implementar diseños de muy diferente tipo desde pequeños sistemas empotrados usando únicamente RVE hasta supercomputadores con aceleradores vectoriales RV64GV; como ejemplo [14] se mencionan algunos proyectos recientes:

- Pequeños sistemas integrados inteligentes con tecnologías microelectrónicas como fue el proyecto “Lagarto” liderado por Barcelona Supercomputing Center (BSC), considerado el primer chip con ISA de código abierto desarrollado en España en el año 2019.
- Desarrollo de tres chips para ámbitos específicos de aplicación (seguridad, criptografía, medicina, conducción automática de vehículos, etc.) como es el caso del iniciado proyecto DRAC (Designing RISC-V-based Accelerators for next generation Computers) en el año 2020.
- El nuevo supercomputador, MareNostrum V, está siendo diseñado en Barcelona [15], procesará 200.000 billones de operaciones por segundo y contribuirá a no tener que comprar el hardware de los futuros superordenadores, a partir de 2025.

Además, se puede desarrollar software para futuros diseños gracias a que el repertorio, visto a lo largo del presente Capítulo, permite desarrollar código para esta arquitectura tal y como se verá en el próximo Capítulo 3.

Capítulo 3 - Estudio del ecosistema SW

A lo largo de este capítulo se explicarán algunos de los simuladores que se han ido probando en este TFG y herramientas software, como entornos de desarrollo, así como los sistemas operativos capaces de tratar la arquitectura RISC-V. En el ecosistema software no sólo encontramos software libre, sino también hay otro software propietario.

3.1 Simuladores

Dada la gran variedad de simuladores existentes en la arquitectura RISC-V, se ha optado por hacer una selección y centrarse en los más ampliamente utilizados en el ámbito educativo. Por cada simulador, se hará una breve introducción, una tabla en la que figuran las características principales, una explicación de cómo instalarlos, un ejemplo de utilización y figuras donde se muestran las interfaces de los simuladores. Todos los simuladores mostrados a continuación se han probado en un equipo con el sistema operativo Ubuntu 20.04.

3.1.1 RARS

RARS es muy útil para aprender a programar en ensamblador de RISC-V. En la Tabla 3-1 se muestran las características principales del simulador obtenidas en el enlace que aparece en la última fila de la citada tabla.

Simulador	RARS
Plataformas	Linux, macOS, Windows
Licencia	MIT
Soporte	Benjamin Landers
Características	- RV32IMFDN ISA. - Soporte para la depuración utilizando puntos de interrupción.
Enlace	https://github.com/thethirdone/rars

Tabla 3-1. Características del simulador RARS.
Fuente: Elaboración propia a partir de https://github.com/thethirdone/rars.

Para instalar este simulador se deben seguir los siguientes pasos:

1. Entrar en el enlace proporcionado en la Tabla 3-1.
2. Descargar el archivo “rars1_3_1.jar”.
3. Instalar JAVA desde un terminal (sudo apt-getinstall default-jdk)
4. Dar permisos de ejecución al archivo “rars1_3_1.jar”.
5. Hacer doble click sobre él.
6. También se puede arrancar desde el terminal con (java -jar rars1_3_1.jar).

Con estos sencillos pasos el simulador RARS ya está en funcionamiento.

En la interfaz se puede observar, Figura 3-1, un editor de código, una salida de mensajes y el contenido de los registros.

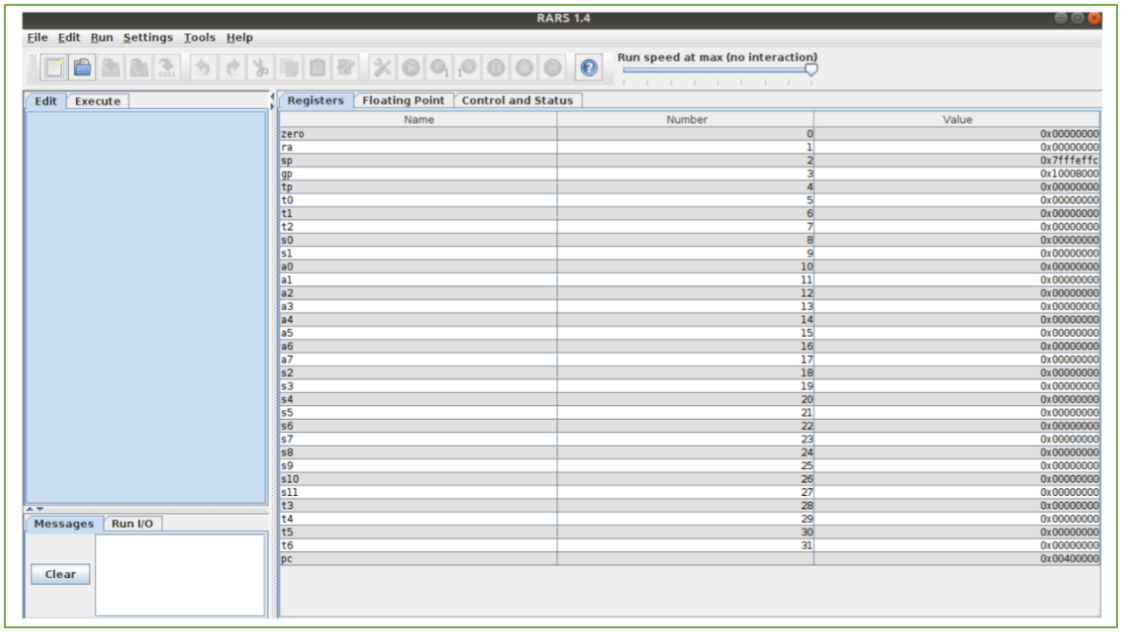

Figura 3-1. Interfaz del simulador.
Fuente: Elaboración propia.

Para realizar la ejecución del programa *“Hola mundo!”*, véase Figura 3-2, lo primero que hay que hacer es escribir el código. Para ello, se debe hacer click en *“file->new”*.

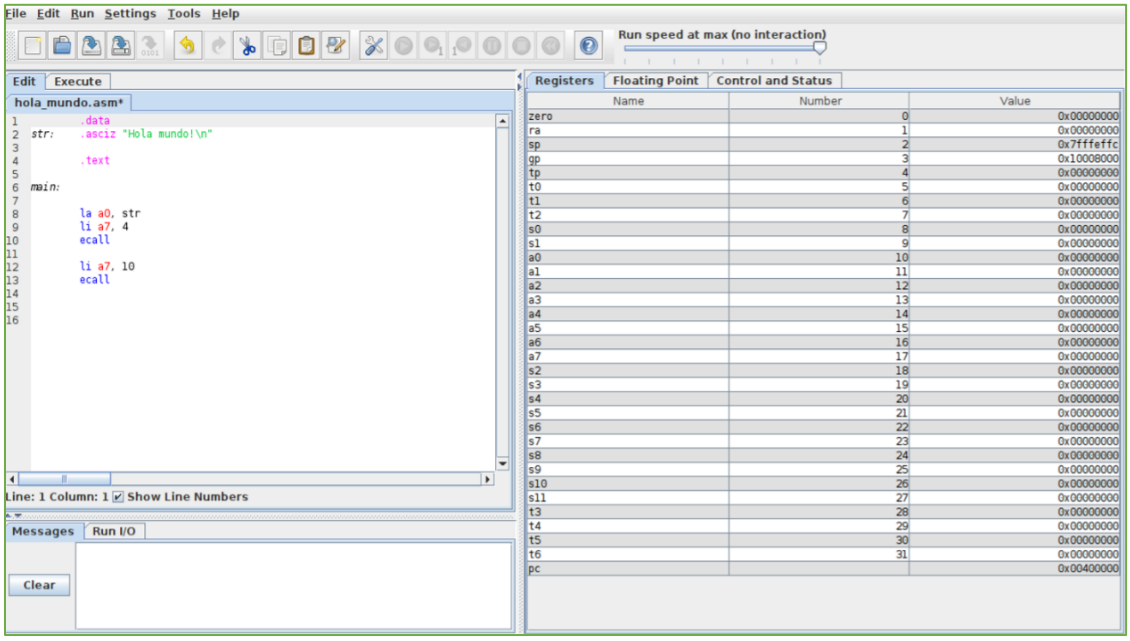

Figura 3-2. Ejecución de "Hola mundo!"
Fuente: Elaboración propia.

A continuación, Figura 3-3, se realiza el ensamblado del programa haciendo click *en "assemble"* (cuadrado señalado en azul en la Figura 3-3), lo que generará el código máquina, código que entiende el procesador y que se almacena en memoria. En la pestaña "execute" se puede ver el contenido de la memoria.

Si el ensamblado se ha realizado correctamente se obtendrá un mensaje en la pestaña de "messages": *"Assemble: operation complete successfully".* En las siguientes figuras, se explicará el significado de los otros cuadros trazados en color de la Figura 3-3:

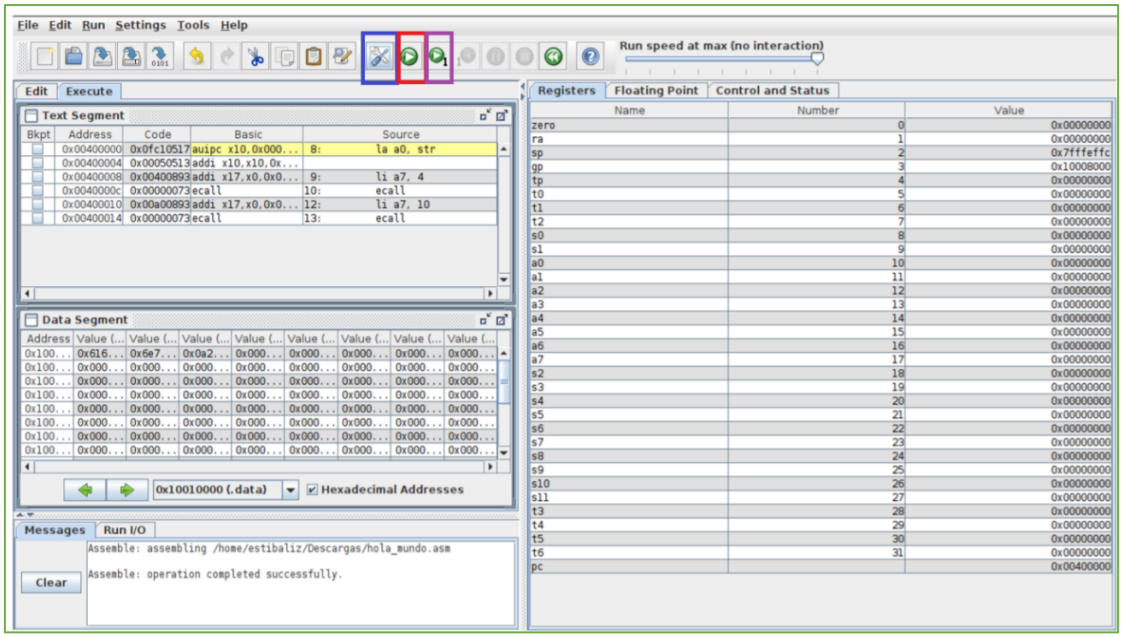

Figura 3-3. Ensamblado del programa.
Fuente: Elaboración propia.

Para realizar la ejecución del programa hay que cliquear en el botón *"Run"* (cuadro señalado en rojo en la Figura 3-3). Una vez ejecutado se puede observar en la salida de mensajes el *"Hola mundo!"* y un mensaje de terminación del programa *"program is finished running",* véase Figura 3-4.

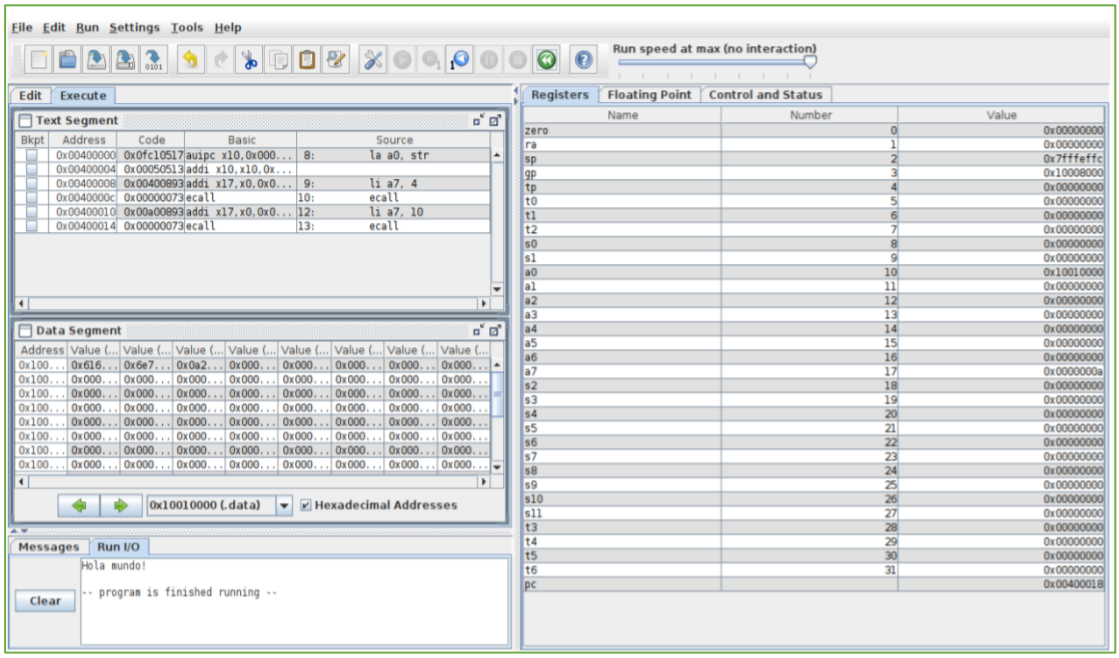

Figura 3-4. Terminación del ensamblado.
Fuente: Elaboración propia.

Este simulador permite ejecutar nuevamente los programas. Para ello, hay que cliquear primero en *"clear"* y segundo en *"reset memory and registers".* Se obtendrá un mensaje *"reset: reset completed",* tal y como se muestra en la Figura 3-5.

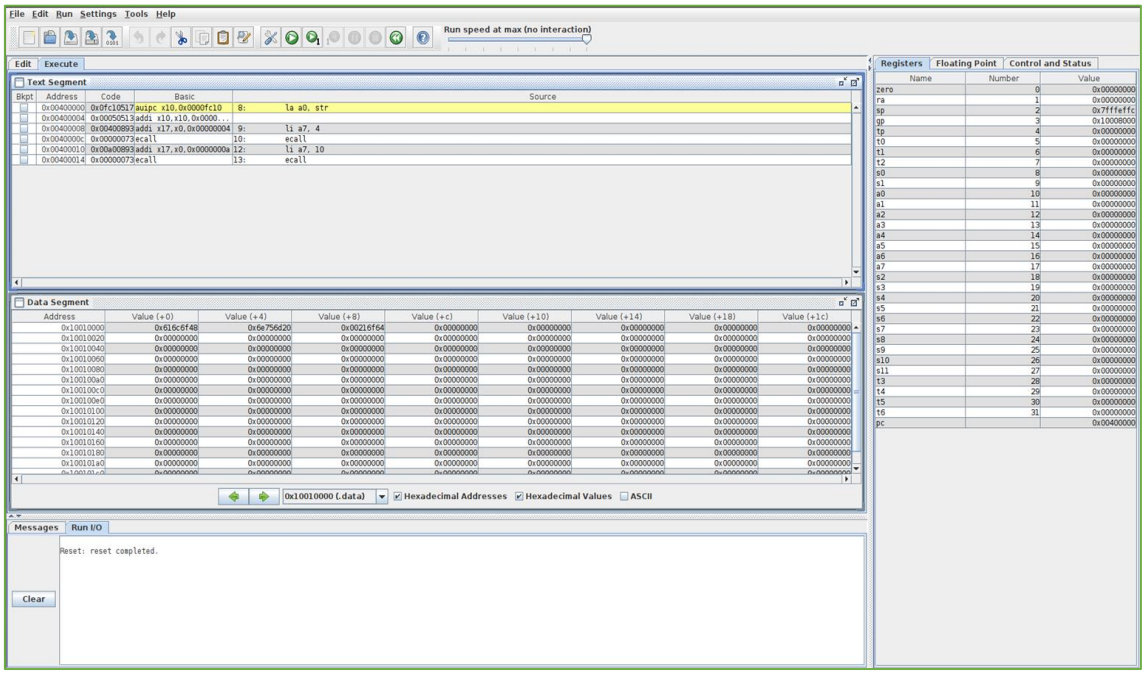

Figura 3-5. Reset del ensamblado.
Fuente: Elaboración propia.

El simulador RARS permite depurar código, paso por paso, cliqueando el cuadro morado mencionado y mostrado en la Figura 3-3, *"Run one step at a time"*. La instrucción que se está ejecutando aparece en amarillo y el registro que se está modificando en verde tal y como se muestra en la Figura 3-6.

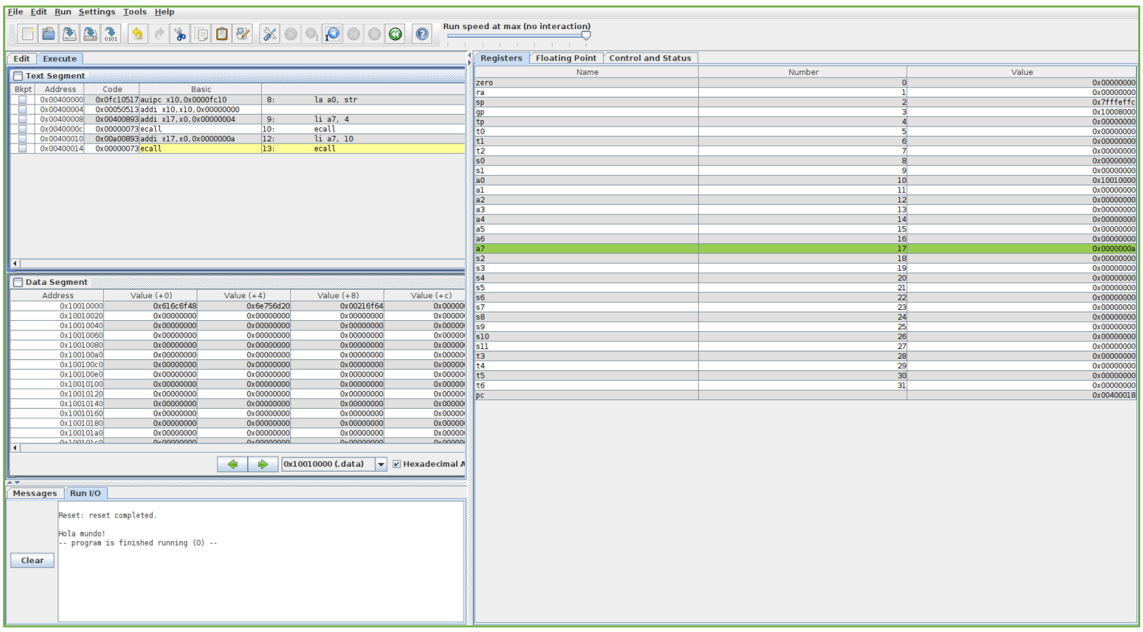

Figura 3-6. Ejecución por pasos.
Fuente: Elaboración propia.

Para finalizar, después de haber procedido al estudio del simulador RARS, se recogen brevemente las siguientes conclusiones:

1. Su instalación resulta sencilla.
2. Para aprender lenguaje ensamblador resulta bastante útil y fácil de usar.
3. En la pestaña settings se pueden añadir o eliminar distintas opciones que permite el simulador, como por ejemplo poner los valores en hexadecimal o que sean de 64 bits.
4. La interfaz es muy intuitiva y a primera vista recuerda al entorno Eclipse que se utiliza en la asignatura de Fundamentos de Computadores II del Grado de Ingeniería Informática UCM.

3.1.2 JUPITER

Jupiter es un simulador del lenguaje ensamblador de RISC-V. Jupiter está orientado al ámbito educativo, y por ello que tiene una interfaz muy fácil de usar, como se explicará a continuación.

Este simulador posee dos interfaces, una gráfica de usuario (modo GUI) que es sobre la que se va a explicar el funcionamiento de la herramienta de forma detallada y otra de línea de comandos (modo CLI). En la Tabla 3-2 se muestran las características básicas de dicho simulador.

Simulador	**Jupiter** 
Plataformas	Linux, macOS, Windows
Licencia	GPL-3.0
Soporte	Andrés Castellanos
Características	- RV32IMF ISA. - Dos modos de operación (interfaz de líneas de comando (CLI) o interfaz gráfica de usuario (GUI). - Muestra comentarios de errores como errores de sintaxis, al intentar acceder a memoria reservada o al intentar escribir en una memoria de solo lectura. - Modular, es decir, ensamblado y simulación de varios archivos a la vez.
Enlace	https://github.com/andrescv/Jupiter

Tabla 3-2. Características del simulador Jupiter.
Fuente: Elaboración propia.

Para preparar este simulador se deben seguir los siguientes pasos:

1. Cliquear el enlace proporcionado en la última fila de la Tabla 3-2.
2. Descargar el archivo "*Jupiter-3.1-linux.zip*".
3. El simulador se puede arrancar de dos maneras: una, en modo CLI *"./image/bin/jupiter [options] <files>"* como aparece en la Figura 3-7 y la otra, en modo GUI mostrada en la Figura 3-8.

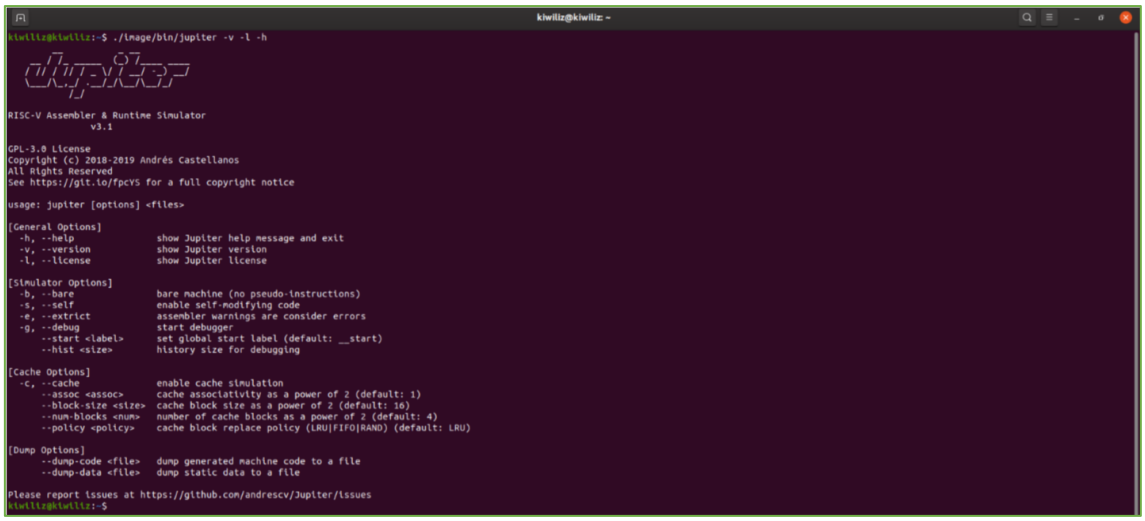

Figura 3-8. Interfaz del modo CLI del simulador.
Fuente: Elaboración propia.

Para usar la interfaz gráfica de usuario en modo GUI hay que ejecutar *"./image/bin/jupiter"*. Con estos pasos el simulador queda instalado y listo para ser ejecutado. La interfaz también resulta muy intuitiva, en la *"zona editor"* se escribe el código del programa, en la *"zona consola"* sale por pantalla el resultado del código.

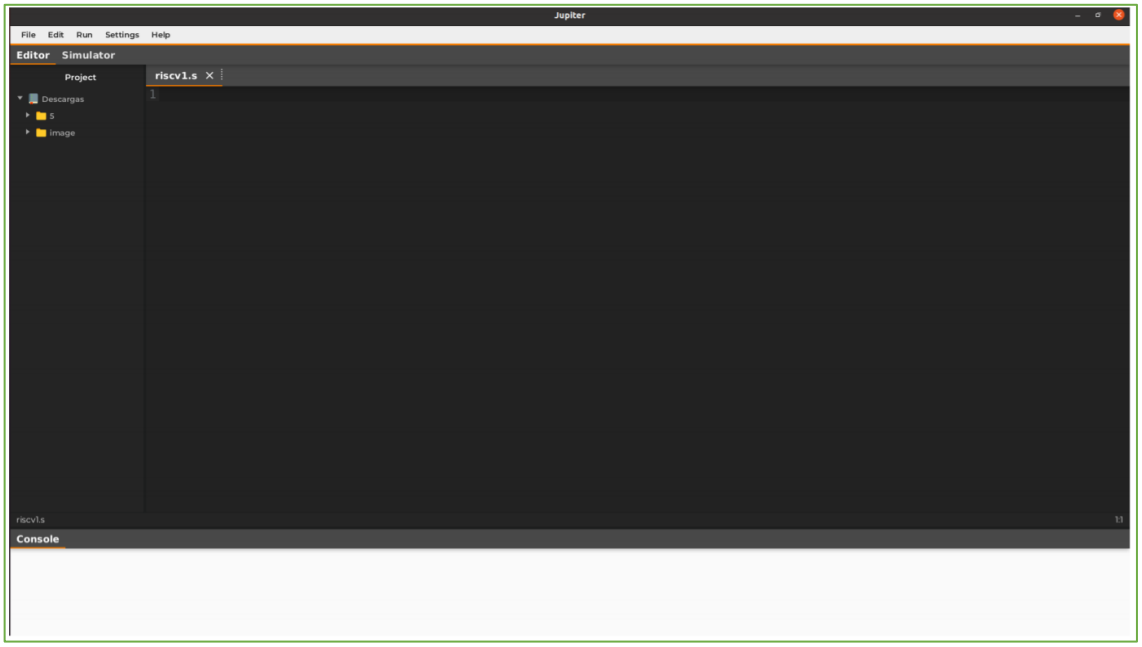

Figura 3-7. Interfaz del modo GUI del simulador.
Fuente: Elaboración propia.

Para hacer funcionar el simulador, lo primero que se debe hacer es escribir el código. Nuevamente, se escribe un *"Hola Mundo!"* (Figura 3-9).

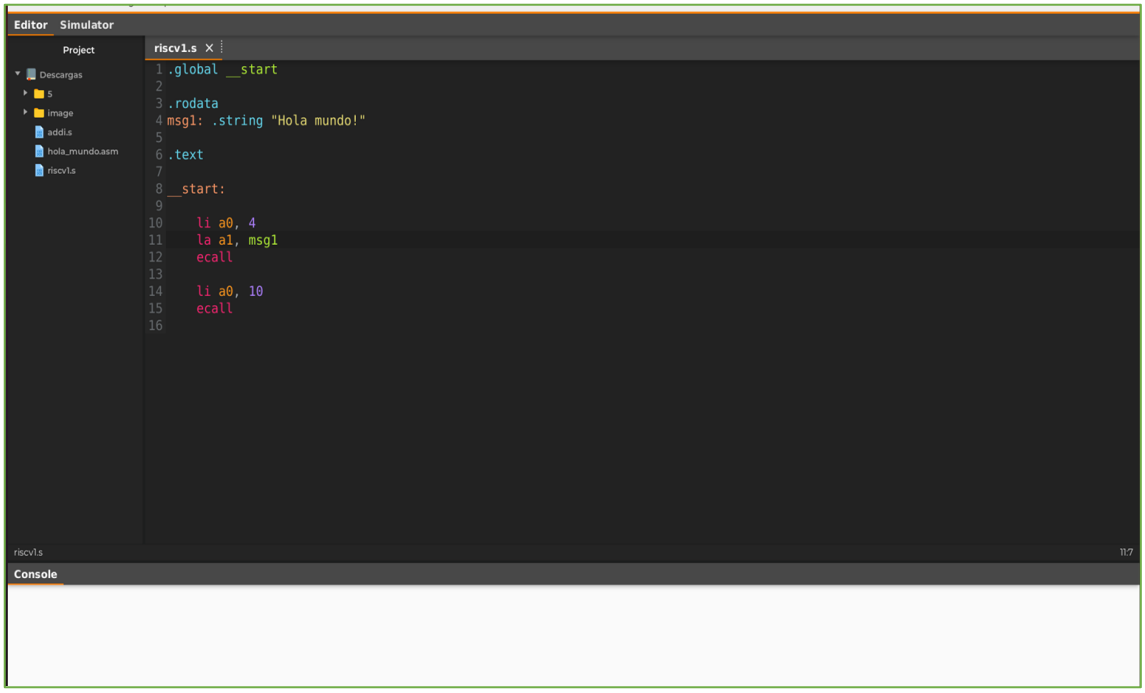

Figura 3-9. Código "Hola mundo!" en simulador Jupiter.
Fuente: Elaboración propia.

A continuación, se realizará el ensamblado. Para ello, vamos a la pestaña *"run – assemble (F3)",* (Figura 3-10).

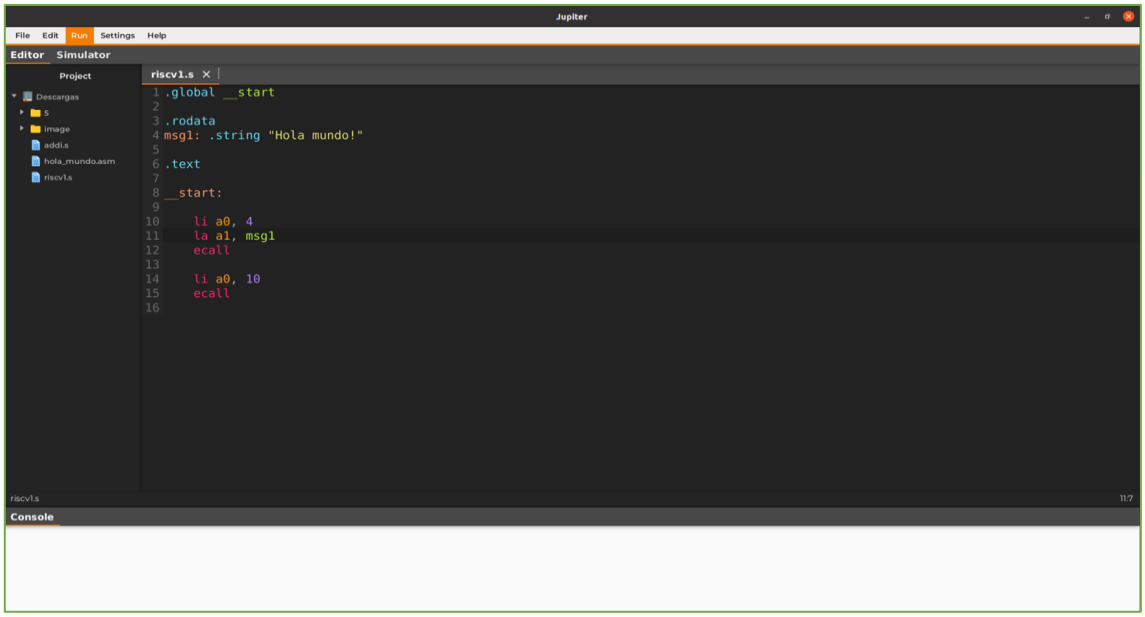

Figura 3-10. Inicio de la ejecución del "Hola Mundo!"
Fuente: Elaboración propia.

Cuando se realiza el ensamblado, el programa aparece tal y como se muestra en la Figura 3-11. Los botones que resultan interesantes son el morado y el azul oscuro. El morado guarda en un archivo .txt el código máquina y el azul oscuro el contenido de la memoria.

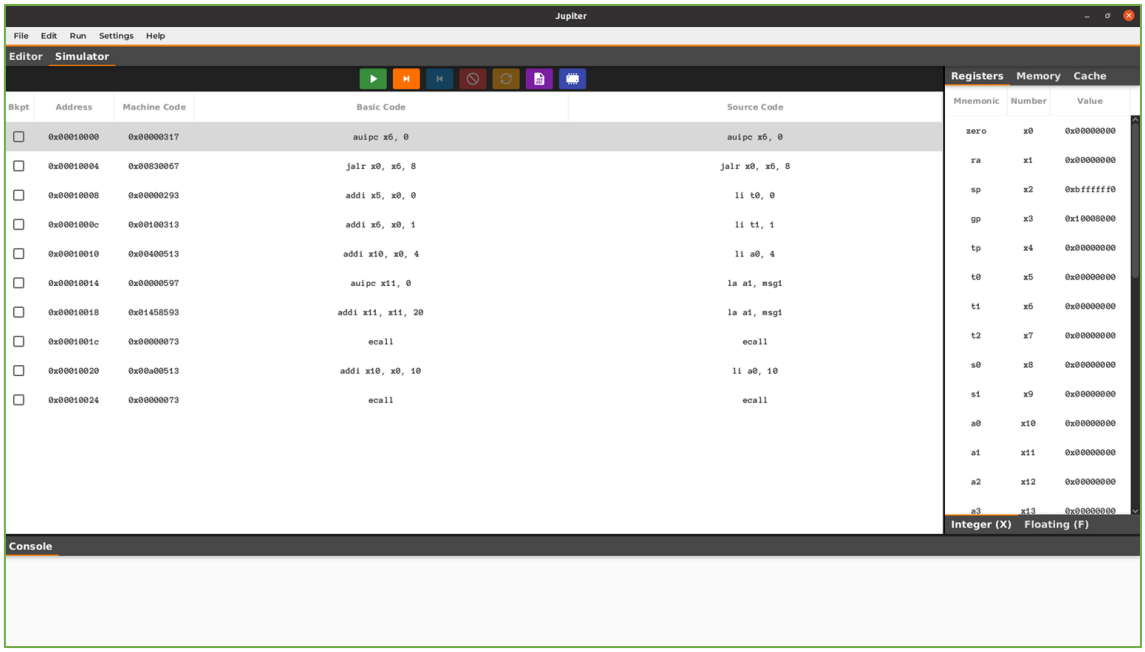

Figura 3-11. Ensamblado del programa.
Fuente: Elaboración propia.

Si le damos al *Play* (botón verde) se realiza la ejecución del programa y se verá en resultado en la Figura 3-12.

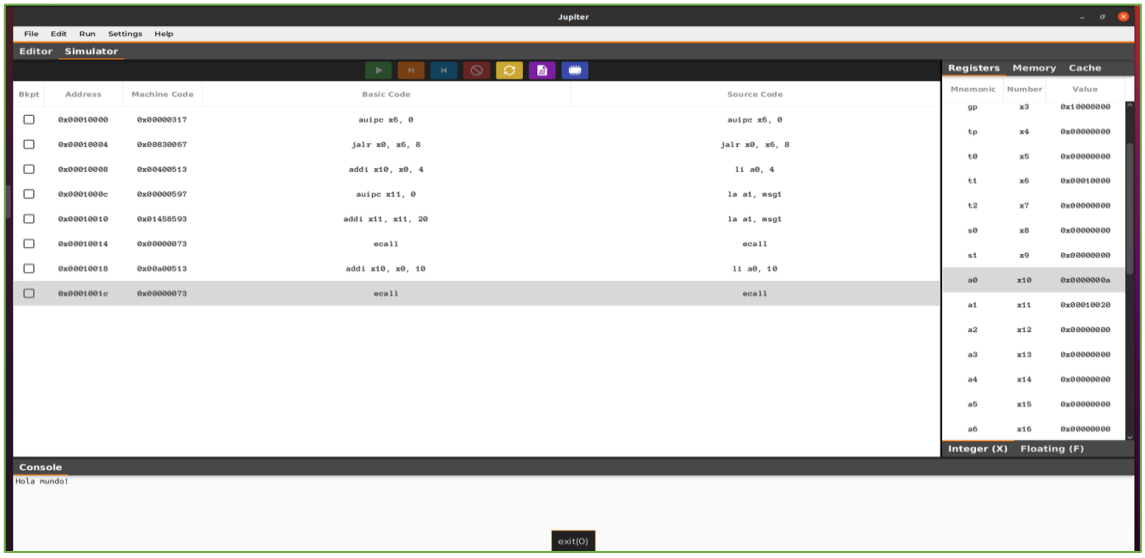

Figura 3-12. Ejecución paso a paso del programa.
Fuente: Elaboración propia.

En este simulador es posible visualizar el contenido de la memoria caché. Para ello hay un botón en la esquina superior derecha *Caché.* Esto es interesante porque aparecen qué marcos de bloques han sido utilizados y cuáles están libres, tal y como se puede ver en la parte derecha de la Figura 3-13.

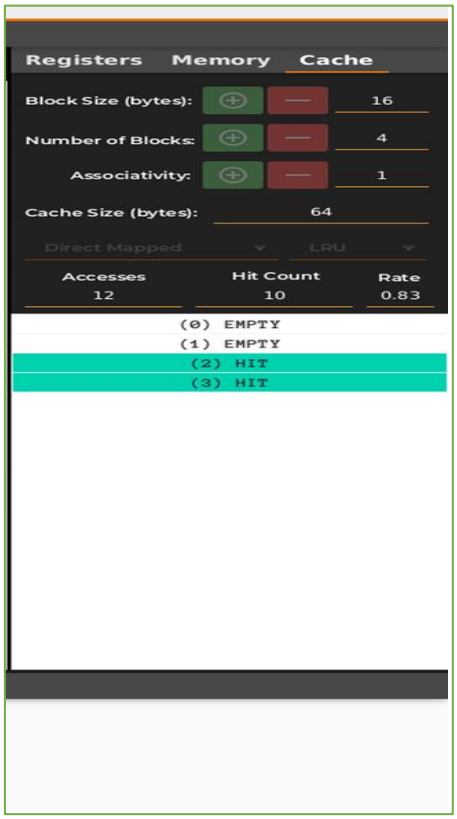

Figura 3-13. Vista de la memoria caché.
Fuente: Elaboración propia.

Para finalizar, considero que este simulador es útil para la asignatura Estructura de Computadores puesto que indica cuántos accesos se realizan a la memoria caché. Además, es capaz de simular todas las instrucciones RV32IMF en modo usuario.

3.1.3 RIPES

Ripes es un simulador del repertorio RISC-V que muestra la ruta de datos asociada a un pipeline de 5 etapas. Se puede usar para saber cómo se ejecuta el código máquina en una variedad de micro-arquitecturas, cómo los diferentes diseños de caché influyen en el rendimiento y cómo se genera código ejecutable a partir del código ensamblador.

Simulador	Ripes
Plataformas	Linux, Windows
Licencia	MIT
Soporte	Morten Borup Petersen
Características	- RV32I - Pipeline RISC de 5 etapas - Ejecución de código a nivel de ensamblador en un procesador RISC-V
Enlace	https://github.com/mortbopet/Ripes

Tabla 3-3. Características Ripes.
Fuente: Elaboración propia.

Para instalar Ripes tenemos que seguir los siguientes pasos:

1. Cliquear en el enlace https://github.com/mortbopet/Ripes.
2. Descargar el archivo de la imagen "*ripes*"
3. Dar permisos de ejecución al archivo y hacer doble click sobre él.
4. Así, Ripes estará listo para ser usado.

En esta ocasión, se usa como ejemplo de funcionamiento la ejecución de la instrucción un *addi x1, x2, 3* que suma, como se muestra en la Figura 3-14. Se ha utilizado este ejemplo ya que el resultado se podrá ver de manera clara. Primero se escribe el código en el apartado *"source code"* y según se va escribiendo código el simulador automáticamente lo traduce a código máquina, con sus direcciones de memoria y las instrucciones que luego ejecuta la CPU, tal y como aparece en la Figura 3-14.

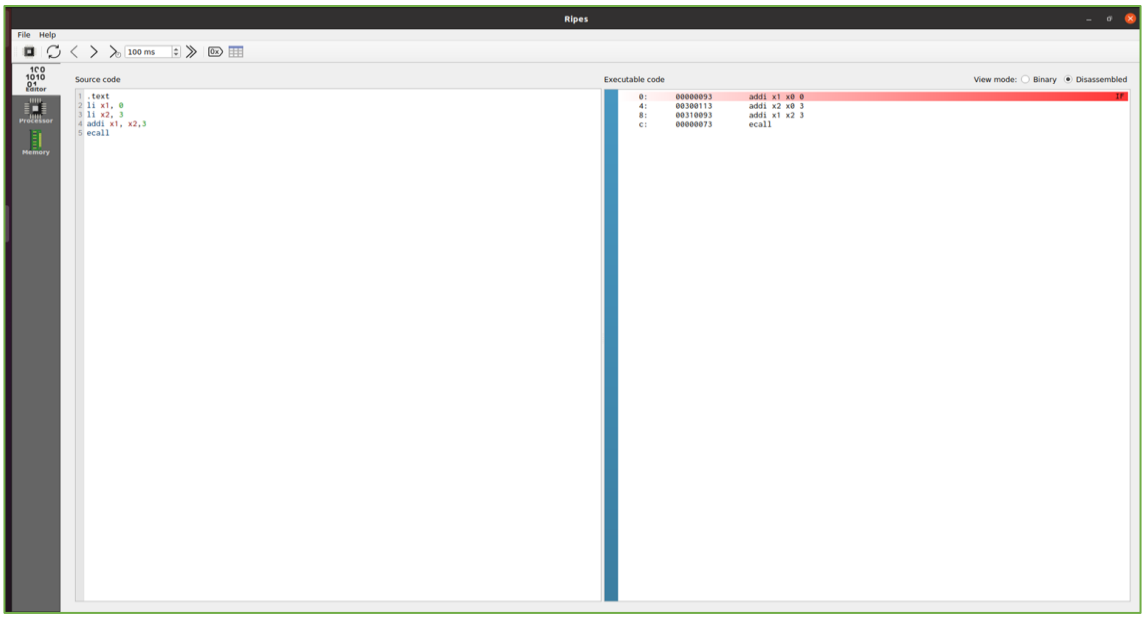

Figura 3-14. Interfaz del simulador.
Fuente: Elaboración propia.

A continuación, Figura 3-15, se puede ver en la esquina superior izquierda un botón *"Processor"*. Si se hace click sobre él, aparece una ruta de datos RISC-V de 5 etapas. Esta ruta resulta muy interesante puesto que aparecen todas las componentes del procesador y se puede ir ejecutando paso por paso observando cómo va cambiando instrucción a instrucción.

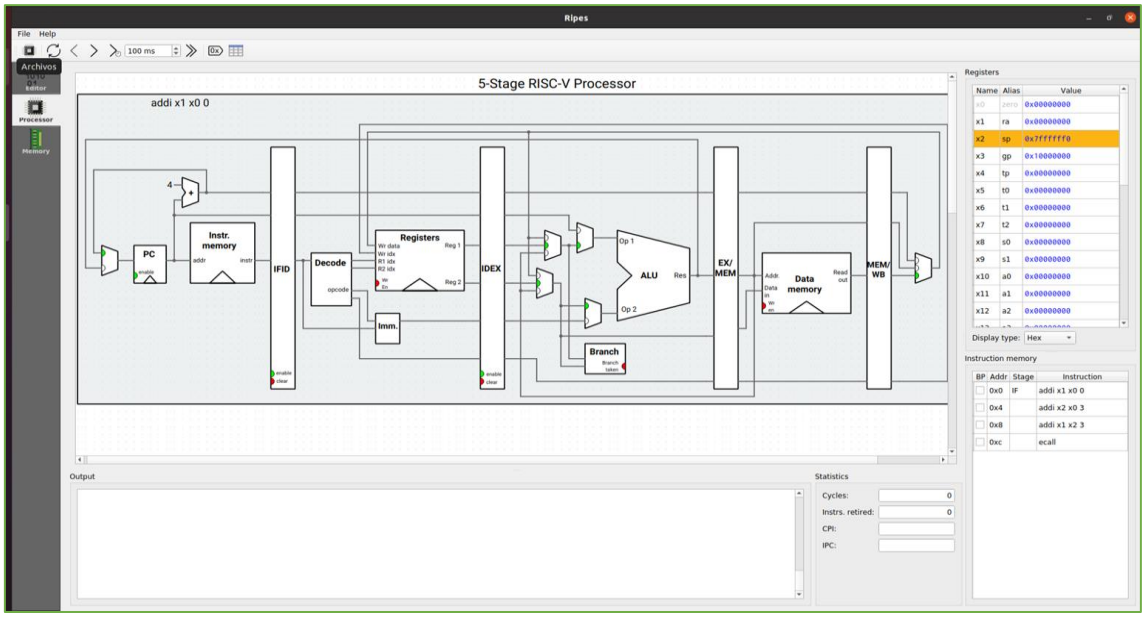

Figura 3-15. Ruta de datos.
Fuente: Elaboración propia.

Una vez ejecutada cada instrucción aparece el resultado (Figura 3-16). Se puede ver como el registro x1 contiene el valor esperado y otras estadísticas que se muestran en la parte inferior de dicha figura, como por ejemplo los ciclos necesarios para la ejecución de la instrucción.

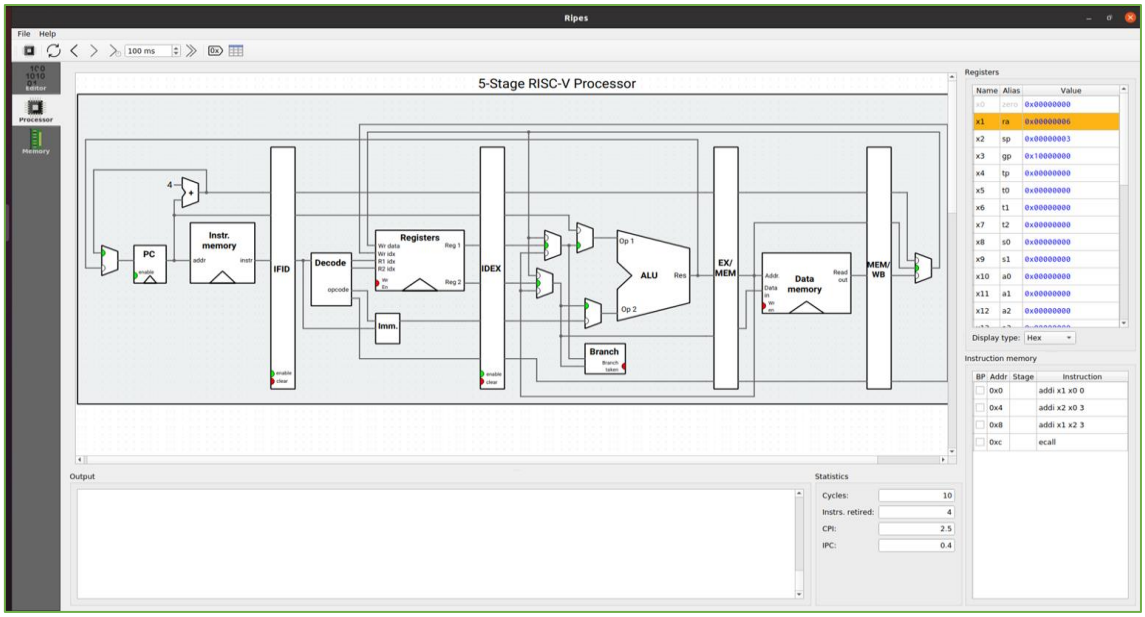

Figura 3-16. Ejecución en la ruta de datos.
Fuente: Elaboración propia.

También se puede ver el contenido de la memoria en el botón de la izquierda *"memory"* (Figura 3-17).

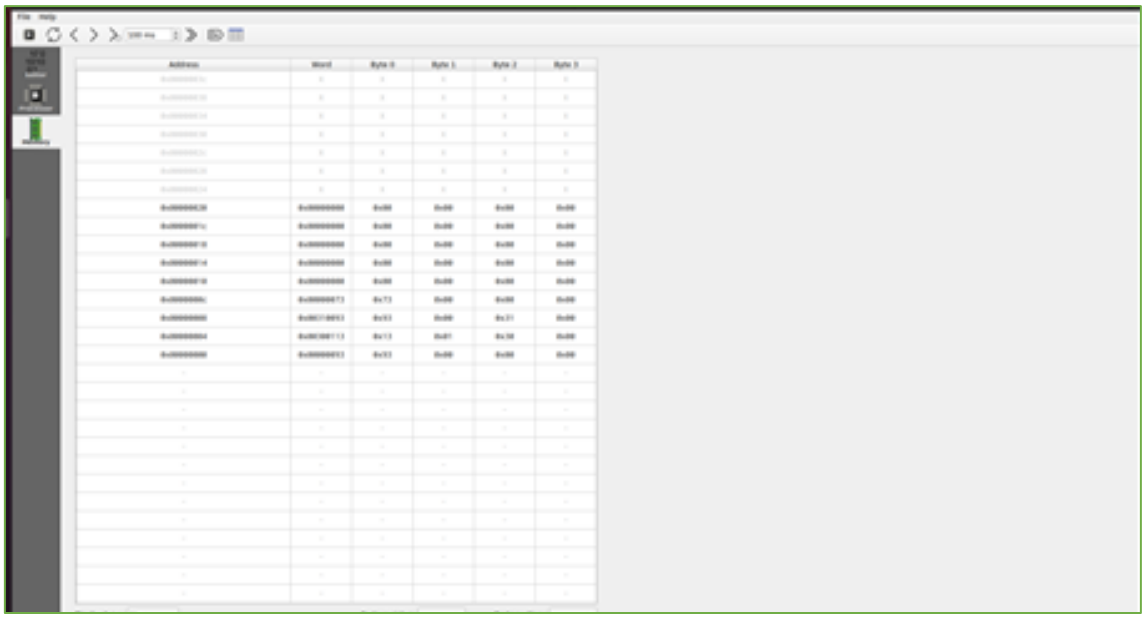

Figura 3-17. Contenido de la memoria.
Fuente: Elaboración propia.

Concluyendo, este simulador es útil para abordar el estudio de las rutas de datos que se diseñan en la asignatura de Fundamentos de Computadores II. Además, se puede ver paso por paso cómo se va ejecutando cada etapa de la instrucción.

Todos los simuladores mostrados anteriormente son fáciles de instalar gracias a la documentación proporcionada. La interacción con el usuario en los tres es muy intuitiva y existe información suficiente de ayuda en caso de necesitarla. Por último, según la utilidad que se les quiera dar se recomienda uno, como se ha mencionado al final de cada apartado del simulador correspondiente.

Además de simuladores, existen otras herramientas software dentro del ecosistema RISC-V como se verá en el siguiente apartado.

3.2 Otras herramientas software

Para probar la arquitectura RISC-V es necesario tener herramientas software adecuadas. Existen depuradores, distribuciones de Linux, Sistemas Operativos de tiempo real y entornos de desarrollo (IDEs) que soportan esta arquitectura. Las herramientas se nombrarán a continuación.

Respecto a las herramientas se muestran las más importantes y conocidas. Para una mayor facilidad de lectura, la información [16] se mostrará en tablas siguientes, al tiempo que se proporcionarán los enlaces externos para poder profundizar en la misma.

En la Tabla 3-4 se muestran los depuradores más importantes:

Nombre	Enlace	Licencia	Soporte
GDB	Upstream	GPLv2	Andrew Burgess (Embecosm), Palmer Dabbelt (Google)
OpenOCD	Upstream repo, Risc-V repo	GPL v2	Tim Newsome, Megan Wachs, Palmer Dabbelt (Google)
GNU MCU Eclipse OpenOCD	Website, Github	GLPv2	Liviu Ionescu
Ozone the J-Link Debugger	Website	SEGGER commercial license (J-Link PLUS)	SEGGER
Imperas Multi Processor Debugger	Website	Imperas Commercial License	Imperas
PlatformIO Unified Debugger	Docs	Apache 2.0	PlatformIO

Tabla 3-4. Depuradores.
Fuente: Elaboración propia.

Las distribuciones Linux más conocidas, a día de hoy, compatibles con RISC-V se muestran en la Tabla 3-5:

Nombre	Enlaces	Soporte
Fedora	fedoraproject.org	Richard WM Jones, Stefan O'Rear, David Abdurachmanov
Debian	debian wiki, mit.edu, riscv.org, Annc	Manuel A. FernandezMontecelo
OpenMandriva	openmandriva.org,openmandriva.org	Bernhard "BeroRosenkränzer
openSUSE	opensuse.org	Andreas Schwab (SUSE)

Tabla 3-5. Distribuciones Linux.
Fuente: Elaboración propia.

En la Tabla 3-6 se muestran los Sistemas Operativos en tiempo real más conocidos compatibles con sistemas RISC-V:

Nombre	Enlaces	Licencia	Soporte
embOS	website, RISC-V port	SEGGER comercial license, free for non-commercial use	SEGGER
RTEMS	rtems.org, docs.rtems.org		Hesham Almatary
FreeRTOS	sourceforge, freetos.org	MIT	AWS
Zephyr	github, docs	Apache 2.0	Karol Gugala (Antmicro), Peter Gielda (Antmicro), Nathaniel Graff (SiFive)

Tabla 3-6. Sistemas operativos en tiempo real.
Fuente: Elaboración propia.

Los entonos de desarrollo *(IDEs)* más populares son los mostrados en la Tabla 3-7. En el Capítulo 5- Casos de uso, los que se utilizarán serán PlatformIO y Freedom Studio. Por ello, se explicarán a continuación.

Nombre	Enlaces	Licencias	Soporte
GNU MCU Eclipse	Website, Repositories,Binarydistribution	EPL-1.0 / various	Liviu Ionescu
Embedded Studio	Website, RISC-V spec	SEGGER commercial license, free for noncommercial use	SEGGER
PlatformIO	Website, IDE, Docs	Apache 2.0	PlatformIO
Freedom Studio	Website	EPL 1.0/various	SiFive

Tabla 3-7. Entornos de desarrollo.
Fuente: Elaboración propia.

PlatformIO

PlatformIO es una extensión gratuita de Visual Studio Code, un entorno de programación IDE. Con ella, se pueden desarrollar códigos y probarlos en los sistemas emprotrados que se utilicen. Admite casi cualquier núcleo RISC-V y SDK específico de la plataforma [17].

Freedom Studio

Freedom Studio está construido sobre el Eclipse IDE y empaquetado con una cadena de herramientas prediseñada y proyectos de ejemplo Freedom E SDK. Este IDE es compatible con todas las placas de desarrollo SiFive RISC-V [17] y será el utilizado para este Trabajo de Fin de Grado en el que se utiliza como caso de estudio la placa SparkFun RED-V Thing Plus.

En RISC-V se está viendo cómo el software creado está funcionado ya que existen numerosas actualizaciones constantes. Por tanto, se puede concluir que el ecosistema software es suficiente, aunque está siendo mejorado con actualizaciones.

Para concluir con este apartado sobre el ecosistema software, se hará referencia tanto a los simuladores, como a las herramientas para desarrollar código:

- Simuladores:
    - Existen varios simuladores que se adaptan a las necesidades de las asignaturas básicas, pero no para las avanzadas (segmentación, …).
    - No hay un simulador que sirva para todas, aunque el más completo y con más opciones a la hora de desarrollar código es el simulador RARS.
- Herramientas para desarrollar código:
    - Los depuradores, Sistemas Operativos e IDEs, ya están disponibles, así que se puede generar código para RISC-V. Estas herramientas son útiles porque se realizan continuas actualizaciones.

Capítulo 4 - Estudio del ecosistema hardware

Actualmente, hay varias empresas que diseñan núcleos en silicio o en Verilog con FPGAs que utilizan la arquitectura RISC-V. Hay empresas que diseñan hardware propio y otras, sin embargo, hacen hardware que es de libre disposición. En este capítulo se exponen las compañías más conocidas y algunas de las placas diseñadas hasta el momento.

4.1 Compañías

Existen varios tipos de compañías, compañías con hardware propietario y otras de hardware de libre disposición. Las más conocidas son Andes, SiFive y Western Digital Corporation.

4.1.1 Andes

Andes es una empresa que se dedica a fabricar cores de 32/64 bits que funcionan con la arquitectura RISC-V y son capaces de correr Linux. En 2016, se convirtió en miembro *fundador* de la Fundación RISC-V, y en el año 2020 en miembro *Premier*. Después de muchos años interaccionando con clientes, Andes se ha convertido en una gran empresa dedicada al diseño para los *system on chips* (SoC).

Se centra en el mercado integrado y ofrece núcleos de CPU con un entorno de desarrollo integrado y software y hardware asociados para el desarrollo de SoC. Andes está clasificada como la quinta empresa de propiedad intelectual del mundo.

Ha creado una modificación de la arquitectura junto con instrucciones específicas que incluyen en sus propios procesadores. Andes V5 es una extensión de RISC-V que tiene instrucciones para propósitos específicos como se muestra en la Tabla 4-1 [18].

La SV32 y SV39/48 significa que tiene un coprocesador que es un acelerador vectorial.

Serie	Nombre del núcleo	Especificaciones	Lenguaje	Bits	Propósito
A-Series	A25	RV32GCP + SV32 + Andes V5 ext.	Verilog	32	5G, AI, Networking, Video Surveillance,
A-Series	AX25	RV64GCP + SV39/48 + Andes V5 ext.	Verilog	64	5G, AI, Networking, Video Surveillance
A-Series	A25MP	RV32GCP + SV32 + Andes V5 ext. + Multi-core	Verilog	32	5G, AI, Datacenter, Video Surveillance, Networking
A-Series	AX25MP	RV64GCP + SV39/48 + Andes V5 ext. + Multi-core	Verilog	64	5G, AI, Datacenter, Video Surveillance, Networking
N/D-Series	N22	RV32IMAC/EMAC + Andes V5/V5e ext.	Verilog	32	IoT, Sensing, Audio, GPS
N/D-Series	D25F	RV32GCP + Andes V5 ext	Verilog	32	IoT, Sensing, Audio, GPS
N/D-Series	N25F	RV32GC + Andes V5 ext.	Verilog	32	IoT, Sensing, Audio, GPS
N/D-Series	NX25F	RV64GC + Andes V5 ext.	Verilog	64	IoT, Sensing, Audio, GPS

Tabla 4-1. Núcleos Andes.
Fuente: Elaboración propia.

4.1.2 SiFive

SiFive es una empresa que se dedica al diseño de semiconductores y es proveedor de núcleos que funcionan con la arquitectura RISC-V. Fue la primera empresa que hizo hardware propietario basado en RISC-V. Fue fundada en 2015 por los propios investigadores de la arquitectura, entre ellos, Andrew Waterman. SiFive diseña desde núcleos, SoCs, hasta placas de desarrollo. En SiFive se puede diseñar nuestro propio procesador con las pautas que aparecen en la página web.

SiFive tiene series muy distintas ya que cambian la implementación, por ejemplo, el pipeline E7 tiene 8 etapas y E2 tiene 3 etapas. A pesar de tener diferentes números de

etapas, son muy distintas, aunque todas las series utilizan el mismo repertorio de instrucciones.

Los núcleos [19] basados en RISC-V se muestran en la Tabla 4-2:

Serie	Nombre del núcleo	Especificaciones	Lenguaje	Bits	Propósito
E Cores	E7	RV32IMAFDC 2.2	Verilog	32	MCU, Edge computing, AI, IoT
E Cores	E3	RV32IMAFDC 2.2	Verilog	32	MCU, Edge computing, AI, IoT
E Cores	E2	RV32IMAFDC 2.2	Verilog	32	MCU, Edge computing, AI, IoT
S Cores	S7	RV64GC 2.2	Verilog	64	Storage, Virtual Reality, Machine Learning
S Cores	S5	RV64GC 2.2	Verilog	64	Storage, Virtual Reality, Machine Learning
S Cores	S2	RV64GC 2.2	Verilog	64	Storage, Virtual Reality, Machine Learning
U Cores	U7	RV64GC 2.2	Verilog	64	Linux, Datacenter, Network Baseband
U Cores	U5	RV64GC 2.2	Verilog	64	Linux, Datacenter, Network Baseband

Tabla 4-2. Núcleos SiFive.
Fuente: Elaboración propia.

4.1.3 Western Digital Corporation

Esta empresa es conocida por la fabricación de discos duros, pero también fabrica núcleos basados en RISC-V. Esta empresa es la única de las tres que fabrica hardware open-source, que se pueden obtener de su Github.

Los núcleos fabricados con este tipo de arquitectura son [20]:

Nombre del núcleo	Especificaciones	Lenguaje	Bits	Propósito
SweRV EH1	RV32IMC 2.1	SystemVerilog	32	Storage, IoT, Datacenter
SweRV EH2	RV32IMAC 2.1	SystemVerilog	32	IoT, AI, data-intensive embedded applications
SweRV EL2	RV32IMC 2.1	SystemVerilog	32	IoT, AI, data-intensive edge

Tabla 4-3. Núcleos Western Digital Corporation.
Fuente: Elaboración propia

Se concluye este apartado señalando que las áreas en las que más se están utilizando diseños basados en RISC-V son el sector de videovigilancia, aceleradores para aplicaciones de inteligencia artificial, especialmente redes neuronales, visión artificial y aprendizaje automático.

4.2 SoCs

Algunos de los SoCs [21] basados en RISC-V más conocidos y utilizados en este TFG que se pueden encontrar en el mercado actualmente son los siguientes (Tabla 4-4):

SoC	Proveedor	Núcleo	ISA	Sistema Operativo	DevKit
FE310-G00	SiFive	E31	RV32IMAC	RTOS	HiFive1
FE310-G002	SiFive	E31	RV32IMAC	RTOS	HiFive 1 Rev B
K210	Kendryte	K210	RV64GC	Linux	KD233 development board, Sipeed MAIX/M1 development boards

Tabla 4-4. Características de SoCs.
Fuente: Elaboración propia.

Dado que en este Trabajo de Fin de Grado se realizará posteriormente un tutorial con una placa, se proporcionan a continuación las características de algunas placas cuyo SoC ha sido mencionado anteriormente.

HiFive1

HiFive1 (Figura 4-1) es una placa de desarrollo de bajo coste con Freedom E310 (SoC) [22] y una de las mejores formas de comenzar a crear prototipos y desarrollar aplicaciones RISC-V. Las características principales de la placa son las mostradas en la Tabla 4-5:

Figura 4-1. HiFive1.
Fuente: SiFive. (2020). HiFive1. Obtenido de https://www.sifive.com/boards/hifive1

Microcontrolador	FE310-G00
Tensión de funcionamiento	3,3V y 1,8V
Voltaje de entrada	Conector USB de 5V o 7-12 VCC
Voltajes IO	Compatible con 3,3 o 5V
Pines de E/S digitales	19
Pines PWM	9
Controladores SPI / Pines HW CS	1/3
Pines de interrupción externos	19
Pines de activación externos	1
Memoria flash	128Mbit Off-Chip (ISSI SPI Flash)
Interfaz de host (microUSB)	Programación, depuración y comunicación en serie
Peso	22g

Tabla 4-5. Características de HiFive.
Fuente: SiFive. (2020). HiFive1. Obtenido de https://www.sifive.com/boards/hifive1

HiFive1 Rev B

HiFive Rev B (Figura 4-2) es una actualización de la placa HiFive1 [23], cuyas características principales son las mostradas en la Tabla 4-6:

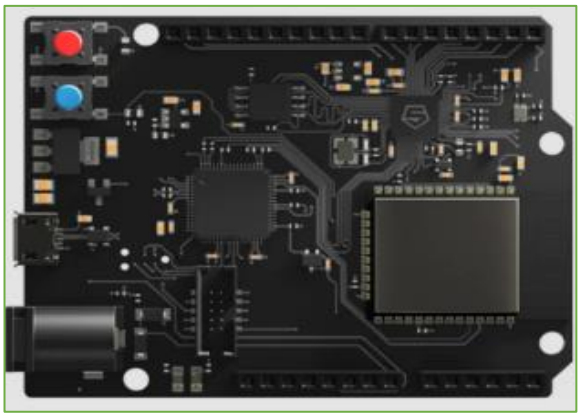

Figura 4-2. HiFive1 Rev B
Fuente: SiFive. (Agosto de 2020). HiFive1 Rev B. Obtenido de https://www.sifive.com/boards/hifive1-rev-b

Microcontrolador	FE310-G02
Tensión de funcionamiento	3,3V y 1,8V
Voltaje de entrada	Conector USB de 5V o 7-12 VCC
Voltajes IO	3,3
Pines de E/S digitales	19
Pines PWM	9
Controladores SPI / Pines HW CS	1/3
Pines de interrupción externos	19
Pines de activación externos	1
Memoria flash	32Mbit Off-Chip (ISSI SPI Flash)
Interfaz de host (microUSB)	Programación, depuración y comunicación en serie
Peso	22g
UART	2
I2C	1
Redes	WiFi / BT (fuera de chip)
Depurar	Segger J-Link, descarga de código drap/drop

Tabla 4-6. Características HiFive1 Rev B.
Fuente: SiFive. (agosto de 2020). HiFive1 Rev B. Obtenido de https://www.sifive.com/boards/hifive1-rev-b

HiFiveUnleashed

HiFiveUnleashed (Figura 4-3) es el último modelo de las placas actuales de SiFive, con Freedom U540, el primer procesador RISC-V multinúcleo con capacidad para Linux [24], cuyas características principales aparecen en la Tabla 4-7:

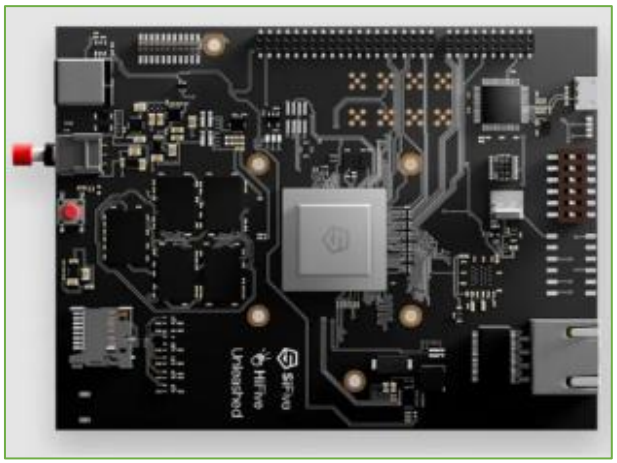

Figura 4-3. HiFive Unleashed.
Fuente: SiFive. (s.d.). HiFive Unleashed. Obtenido de https://www.sifive.com/boards/hifive-unleashed

SoC	SiFiveFreedom U540 SoC
Memoria	8 GB DDR4 con ECC
	Puerto Ethernet Gigabit
Memoria Flash	Flash SPI cuádruple de 32MB de ISSI
Almacenamiento extraible	Tarjeta micro SD
Futura expansión	Conector FMC

Tabla 4-7. Características HiFiveUnleashed.
Fuente: SiFive. (s.d.). HiFive Unleashed. Obtenido de https://www.sifive.com/boards/hifive-unleashed

SparkFun RED-V Thing Plus

Esta es la placa que se ha utilizado en el presente Trabajo de Fin de Grado (Figura 4-4). Esta placa de desarrollo de bajo coste es de Thing Plus, su ISA es RISC-V y tiene un microcontrolador Freedom E310-G002. Las características se muestran en la Tabla 4-8 [25].

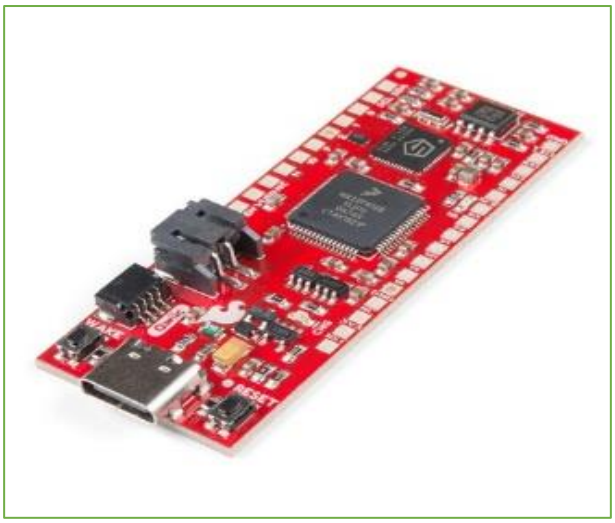

Figura 4-4. SparkFun RED-V Thing Plus
Fuente: SparkFun Electronics. (2020). RED-V Thing Plus Hookup Guide. Obtenido de https://learn.sparkfun.com/tutorials/red-v-thing-plus-hookup-guide

Nombre	Tipo	Proveedor	Lenguaje	Núcleo	ISA	Capacidad del sistema operativo	DevKit
Freedom E310-G002	SoC	SiFive	Chisel	E31	RV32IMAC	RTOS	HiFiveRev B

Tabla 4-8. Características Freedom E310-G002.
Fuente: SiFive. (2020). SiFive FE310-G002 Manual v19p05. Obtenido de https://sifive.cdn.prismic.io/sifive%2F59a1f74e-d918-41c5-b837-3fe01ba7eaa1_fe310-g002-manual-v19p05.pdf

El núcleo que contiene es el E31 y sus características se muestran en la Tabla 4-9:

Nombre	ISA	Caché	Memoria ROM	Memoria RAM	Modos
E31	RV32IMAC	2 vías de 8Kib	ITIM de como máximo 8KiB.	16KiB DTIM	Modo máquina, modo usuario

Tabla 4-9. Características E31.
Fuente: SiFive. (2020). SiFive FE310-G002 Manual v19p05. Obtenido de https://sifive.cdn.prismic.io/sifive%2F59a1f74e-d918-41c5-b837-3fe01ba7eaa1_fe310-g002-manual-v19p05.pdf

La placa SparkFun RED-V Thing Plus [26] posee un microcontrolador SiFive Freedom E310 (FE310-G002) y tiene un núcleo SiFive E31 con la arquitectura RV32IMAC de 32 bits. Permite el modo máquina y modo usuario, la multiplicación y división y las operaciones atómicas e instrucciones comprimidas. Su velocidad es de 150 MHz y su rendimiento de 1,61DMIP/MHz. Tiene una caché de instrucciones de 16 KB.

Las características principales de la SparkFun aparecen en la Tabla 4-10:

Microcontrolador	FE310-G02
Tensión de funcionamiento	3,3V y 1,8V
Voltaje de entrada	Conector USB-C de 5V, conector JST, conector Qwiic
Voltajes IO	3,3
Pines de activación externos	1
Memoria	16KB
Interfaz de host (USB-C)	Programación, depuración y comunicación en serie
Peso	22g
I2C	1

Tabla 4-10. Características SparkFun RED-V Thing Plus.
Fuente: SparkFun Electronics. (2020). RED-V Thing Plus Hookup Guide. Obtenido de https://learn.sparkfun.com/tutorials/red-v-thing-plus-hookup-guide

Para concluir este capítulo sobre el ecosistema hardware, mencionar lo siguiente:

- Hay empresas que están fabricando estos sistemas en determinados mercados.
- Existen placas para sistemas empotrados y alguna puede ejecutar Linux. De momento, no para propósito general.

En el próximo capítulo se expondrán unos tutoriales básicos para el desarrollo de programas tanto en C como en lenguaje ensamblador usando Visual Studio Code y Freedom Studio sobre la placa *SparkFun Red-V Thing Plus*.

Capítulo 5 - Casos de uso de RISC-V

En este capítulo se realiza un tutorial básico sobre cómo hacer un programa en el lenguaje de alto nivel C y ensamblador, para ejecutar en la placa ThingPlus, con la placa conectada al ordenador, usando dos IDE distintos: PlatformIO en Visual Studio Code y Freedom Studio de SiFive. En este capítulo todas las capturas de pantalla que aparecen son de elaboración propia.

5.1 Tutorial para desarrollo de programas en C usando Visual Studio Code

Para la realización de un programa en C necesitamos tener instalado el programa Visual Studio Code.

Una vez instalado, hay que incluir la extensión PlatformIO, la cual dispone de las herramientas que van a ser de utilidad para poder trabajar con la placa Sparkfun. Dentro de las plataformas que soporta PlatformIO se encuentra la arquitectura RISC-V.

Para crear un nuevo proyecto, hay que seguir los siguientes pasos:

1. Cuando estemos en *Quick Access* hay que cliquear en *+ New Project,* esto permite dar el nombre al proyecto,
2. Elegir la placa que vamos a utilizar.
3. Seleccionar el framework. En este caso, se realiza un tutorial para desarrollo de programas en C, para la placa SparkFun Red-V Thing Plus y cuyo framework es Freedom E SDK.
4. A continuación, pulsaremos en *finish*, tal y como se muestra en la Figura 5-1.

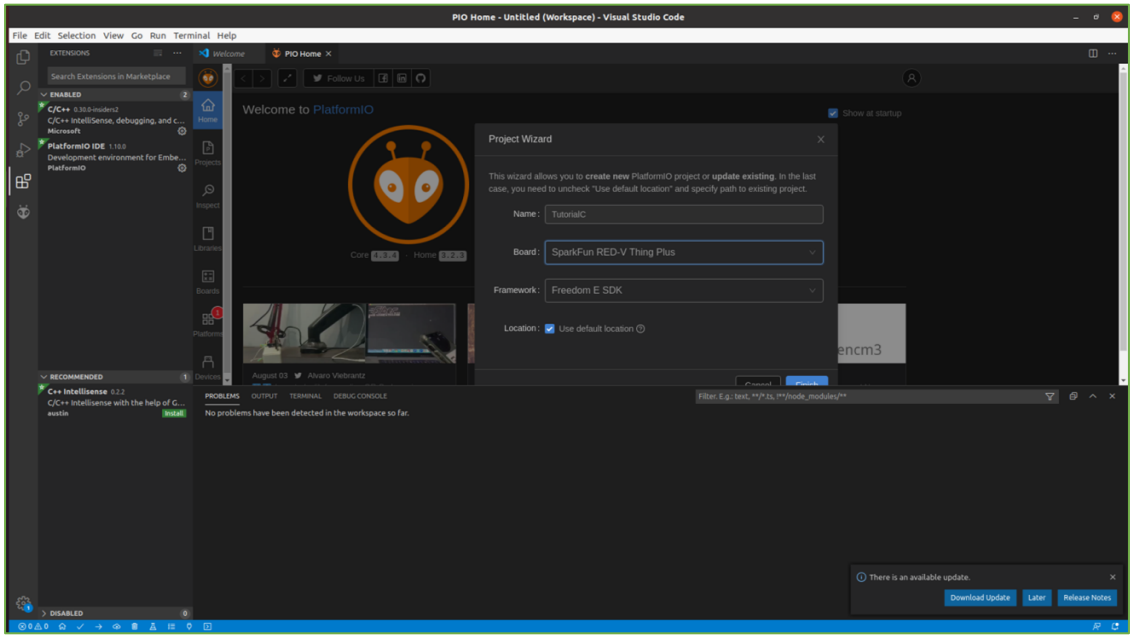

Figura 5-1. Interfaz PlatformIO.
Fuente: Elaboración propia.

Una vez creado el proyecto nos aparece la siguiente pantalla (Figura 5-2). A la izquierda aparecen todos los proyectos creados anteriormente y el proyecto actual. Platformio.ini es un archivo de configuración en el que aparece una configuración por defecto de la placa, la cual se muestra en la Figura 5-2.

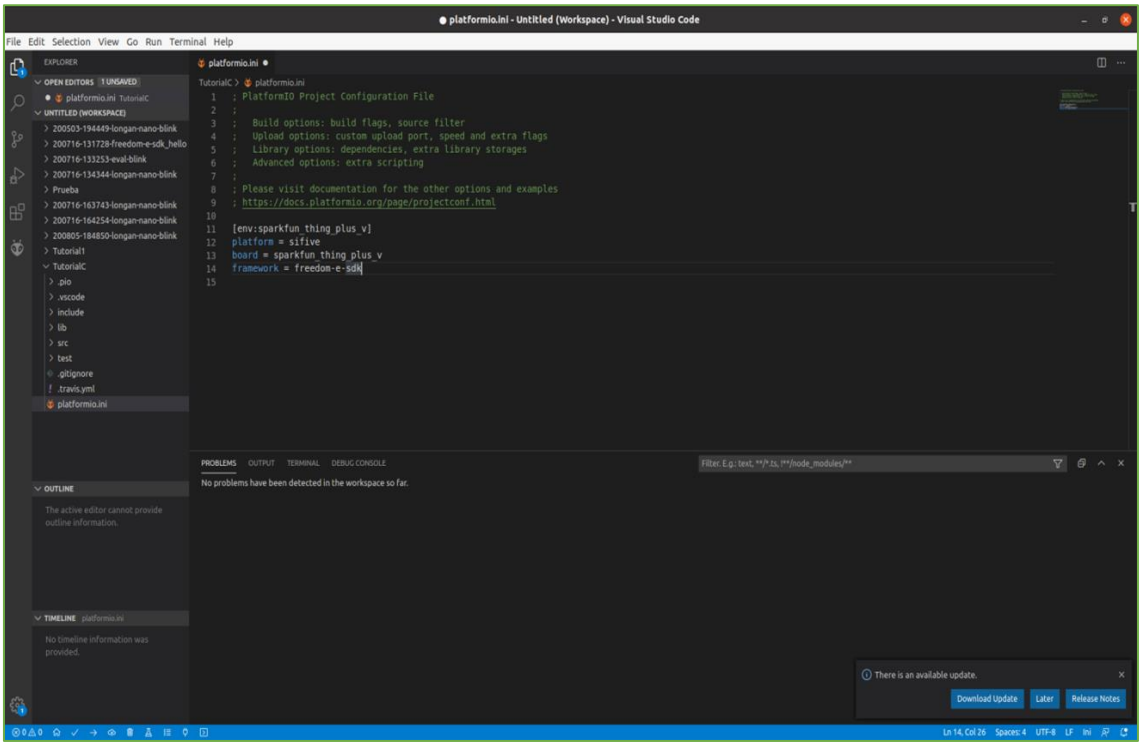

Figura 5-2. Archivo platformio.ini.
Fuente: Elaboración propia.

En la Figura 5-3, se muestra como a esta configuración hay que añadirle *monitor_speed = 115200*. Después, en la parte izquierda en *src* (señalado en rojo) se inserta el fichero con el código en C, que, en este ejemplo, calcula la suma de los elementos de un vector. Para ello, se tiene que cliquear en el botón derecho del ratón, *new file – tutorialC.c*

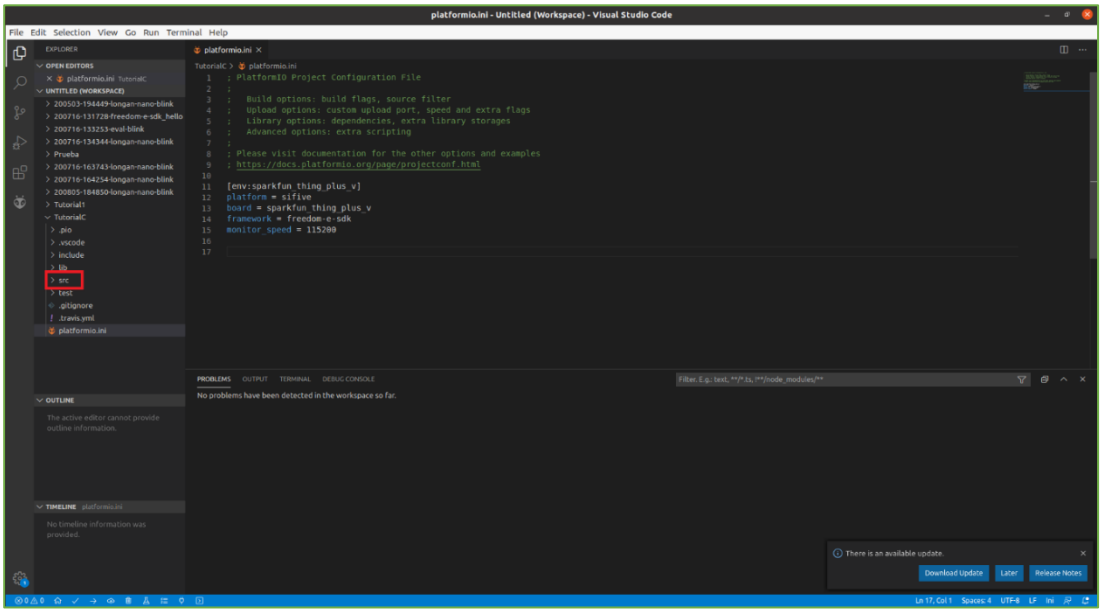

Figura 5-3. Configuración archivo platformio.ini.
Fuente: Elaboración propia.

El código del cálculo de los elementos de la suma de un vector es el mostrado en la Figura 5-4. En el código se añade un “delay” para que dé tiempo a ver el resultado por pantalla ya que es un programa muy rápido de ejecutar.

```
#include <stdio.h>

// To use time library of C
#include <time.h>
void delay(int number_of_seconds)
{
// Converting time into milli_seconds
int milli_seconds = 1000 * number_of_seconds;
// Storing start time
clock_t start_time = clock();
// looping till required time is not achieved
while (clock() < start_time + milli_seconds)
;
}

int sumaV(int v1[], int tam){
int acumula = 0;
int i;

for(i = 0; i < tam; ++i){
acumula += v1[i];
}

return acumula;
}

int main(){
const int tam = 5;
int v1[] = {3, 6, 8, 9, 20};
int resultado = 0;

resultado = sumaV(v1, tam);
delay(3000);
printf(" La suma del vector es: %d\n", resultado);

return 0;

}
```

Figura 5-4. Código suma de los elementos de un vector en C.
Fuente: Elaboración propia.

A continuación, en la Figura 5-5, se muestra en PioHome, menú donde aparecen las posibles acciones a realizar. Se realiza click en *Build*, para compilar el programa que se acaba de crear. Después se cliquea en *Upload and Monitor,* lo cual hace que se vuelque el programa a la placa y se muestre por pantalla el resultado.

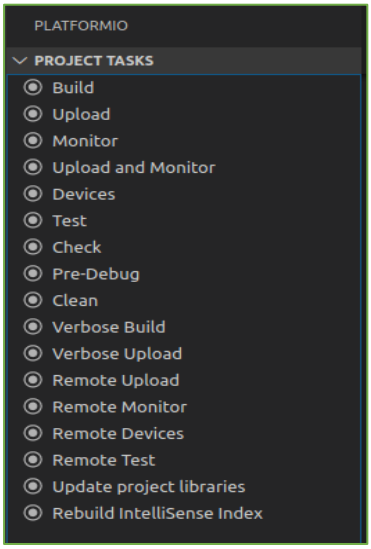

Figura 5-5. PioHome.
Fuente: Elaboración propia.

Una vez que se ha compilado el programa, hay que elegir el puerto. Se procederá a seleccionar 1 (cuadro rojo) y a continuación, se obtendrá el resultado de la suma del vector, 46 (cuadro verde), tal y como se muestra en la Figura 5-6.

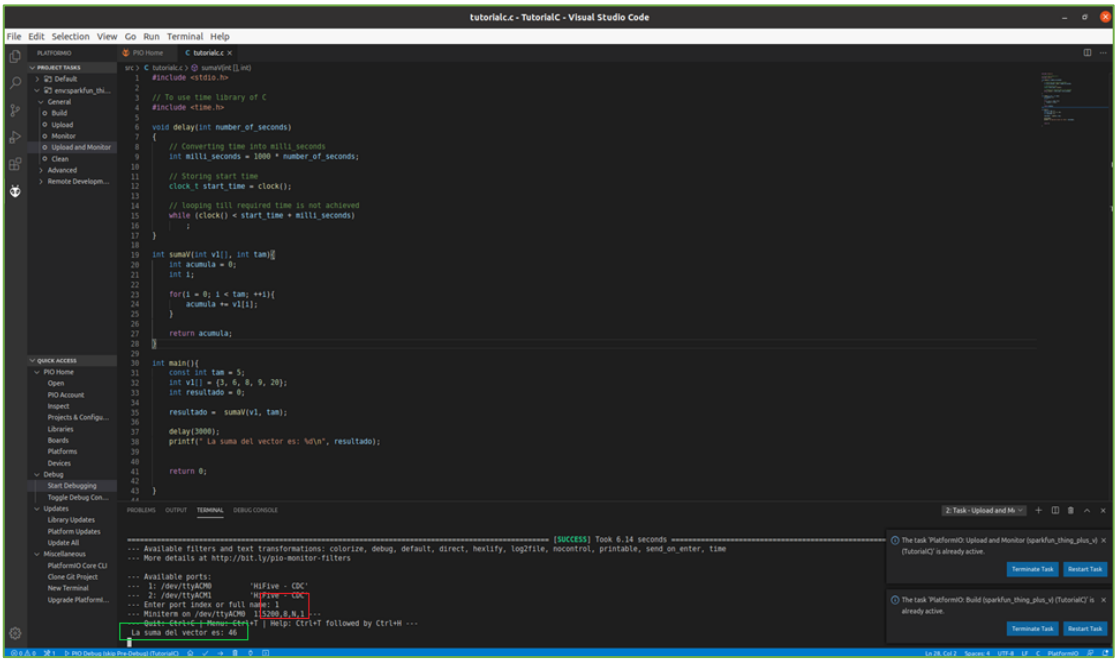

Figura 5-6. Ejecución suma de los elementos de un vector en C.
Fuente: Elaboración propia.

En la documentación del tutorial utilizando Freedom Studio, IDE que se verá posteriormente, se realizará un ejemplo que consiste en apagar y encender el led azul de la placa.

Para probar dicho ejemplo en el Visual Studio, hay que modificar el código previamente; lo primero que hay que hacer es crear un proyecto nuevo, siguiendo los pasos del ejemplo anterior, hasta el paso en el que hay que escribir el código.

Para ello, hay que hacer una función *delay*, que consiste en establecer el tiempo de encendido y apagado de un led. Además, para que se encienda y se apague se usa la librería *metal/gpio.h*, la cual permite manipular la entrada/salida, en este caso el encendido y apagado del led. Con la estructura *"metal_gpio *led0"* se puede realizar el encendido y apagado del led como se muestra en el código de la Figura 5-7.

A continuación, la Figura 5-8 muestra el resultado del led encendido y la Figura 5-9, el resultado de la compilación.

```
#include <stdio.h>
#include <time.h>
#include <metal/gpio.h>
void delay(int miliseconds){
int secs = 10 * miliseconds;
clock_t start = clock();
while (clock() < start + secs){}
}
int main () {

struct metal_gpio *led0; //Instancia de GPIO
led0 = metal_gpio_get_device(0); //IC GPIO es el 5
if (led0 == NULL) {
printf("El led es nulo.\n");
return 1;
}
metal_gpio_disable_input(led0, 5); //Los pins empiezan habilitados por lo que hay que desabilitarlos
metal_gpio_enable_output(led0, 5); //GPIO como output y habilitarlo
metal_gpio_disable_pinmux(led0, 5); //Desabilitar cualquier función que tenga ya el led
metal_gpio_set_pin(led0, 5, 1); //Encender el pin (ON)
while (1) {
metal_gpio_set_pin(led0, 5, 0); //Apagar el led (OFF)
delay(2000000); //Delay
metal_gpio_set_pin(led0, 5, 1); //Encender el pin (ON)
delay(2000000); //Delay
}
return 0;
}
```

Figura 5-7. Código encendido/apagado led en C.
Fuente: Elaboración propia.

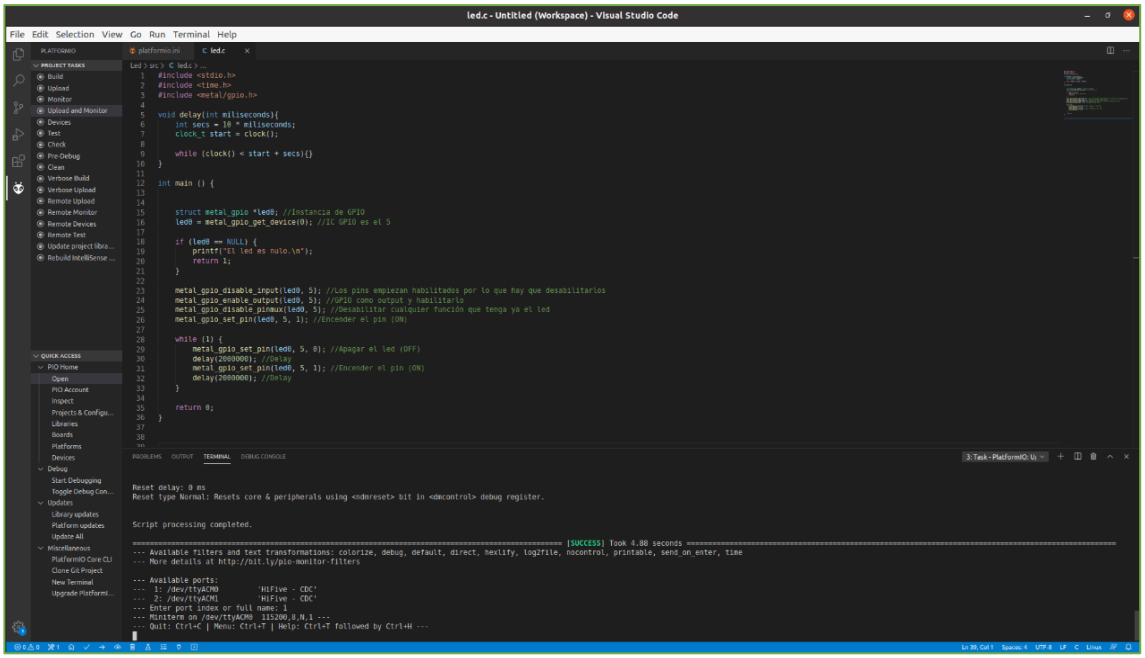

Figura 5-8. Compilación código encendido/apagado led en C.
Fuente: Elaboración propia.

Figura 5-9. Led encendido.
Fuente: Elaboración propia.

En PlatformIO existe un menú de depuración (Figura 5-10). Este menú tiene diversas opciones de ejecución; una de ellas, nos permite pasar de instrucción en intrucción, detener la ejecución del programa o reanudarla.

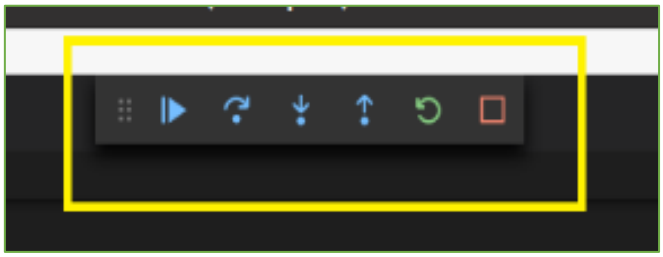

Figura 5-10. Menú depuración PlatformIO.
Fuente: Elaboración propia.

Además, podemos ver el código ensamblador, tal y como se muestra en la Figura 5-11.

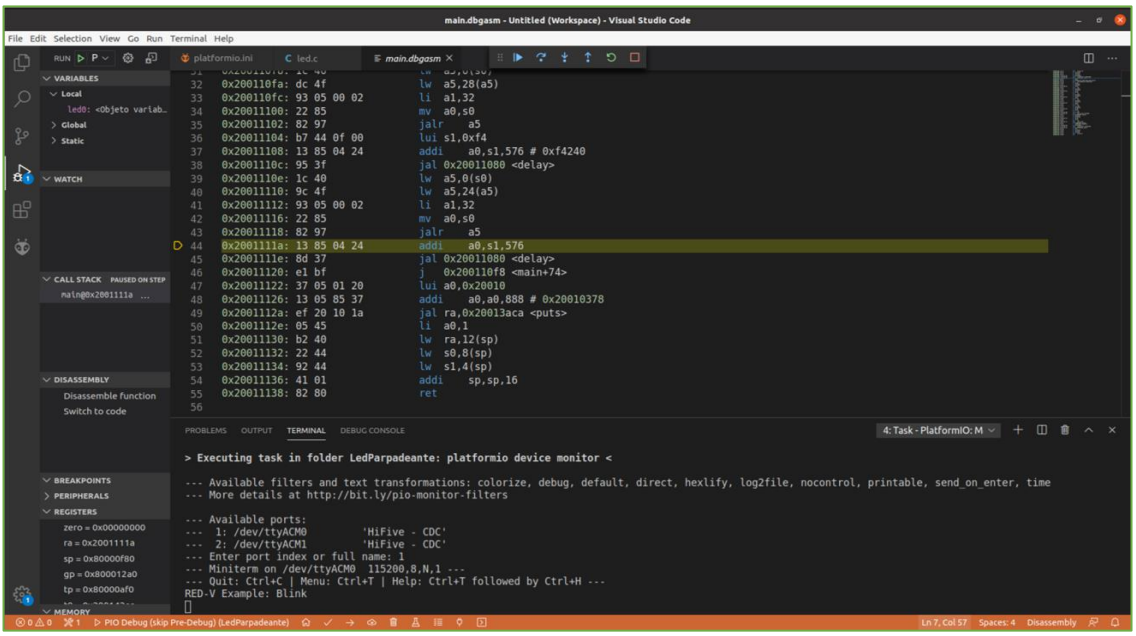

Figura 5-11. Código ensamblador encendido/apagado led.
Fuente: Elaboración propia.

Gracias a la extensión PlatformIO con este IDE resulta sencillo realizar códigos en C, compilarlos, ejecutarlos y volcarlos en la placa. La instalación es muy rápida y su funcionalidad es básica y fácil de entender. Además, existe mucha información en Internet y foros de ayuda.

5.2 Tutorial para desarrollo de programas en C usando Freedom Studio

Para la instalación del programa Freedom Studio debemos seguir los siguientes pasos:

1. Acceder al siguiente enlace: *https://learn.sparkfun.com/tutorials/red-v-thing-plus-hookup-guide* [26] y acceder a RED-V Development Guide.
2. Descargar el archivo que se encuentra en "Download Freedom Studio – v2020.11.0 Linux".
3. Descomprimirlo y guardarlo en una carpeta que no contenga espacios para no tener problemas con el PATH.
4. A continuación, extraer el archivo y después, ejecutar. */FreedomStudio.*

A continuación, se procederá a explicar cómo se crea un proyecto en Freedom Studio mediante el siguiente ejemplo:

1. Crear un proyecto, para ello, hay que ir a File->New->Freedom E SDK Software Project y poner en Select Target la opción sifive-hifive1-revb.
2. Seleccionar el ejemplo que se desea utilizar, en este caso, *hello* tal y como se muestra en la Figura 5-12.

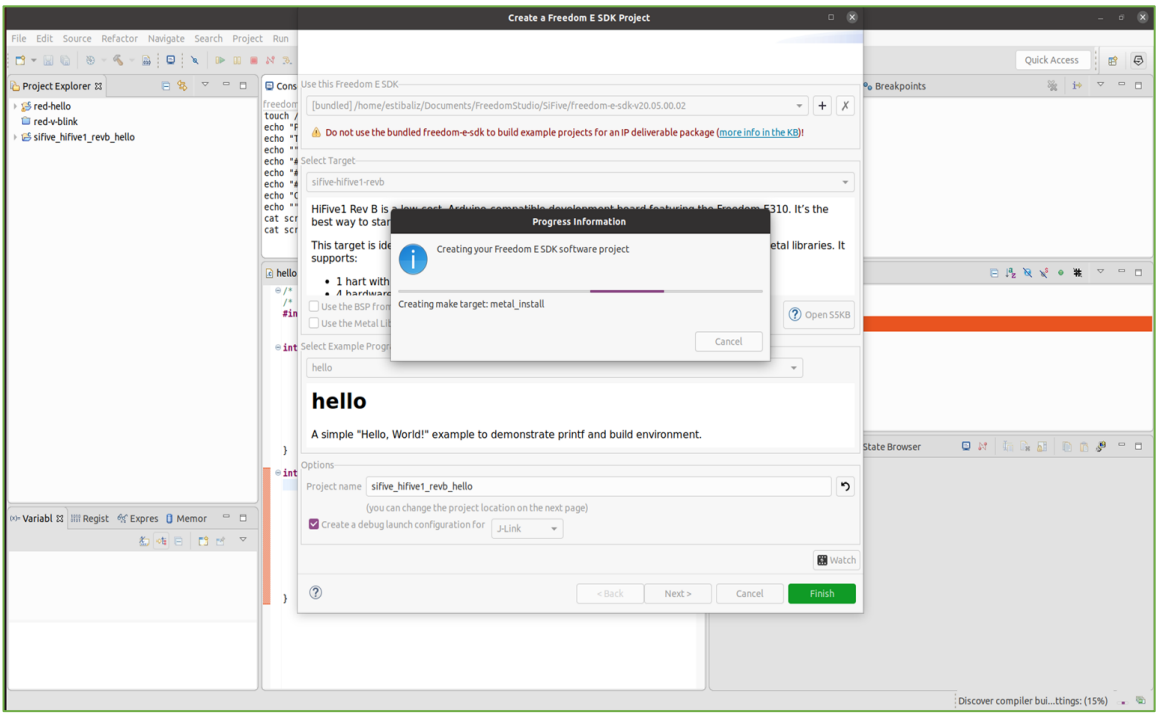

Figura 5-12. Interfaz creación de un proyecto en C.
Fuente: Elaboración propia.

En la Figura 5-13, se muestra el código de ejemplo de *"HelloWorld"*:

```
#include <stdio.h>

int main(){
while(1){
        printf("Hello World\n");
}
return 0;
}
```

Figura 5-13. Código Hello World en C.
Fuente: Elaboración propia.

Seguidamente, Figura 5-14, para ejecutar el código hay que ir a *run->run configurations* y seleccionar el proyecto sobre el que queremos hacer la ejecución.

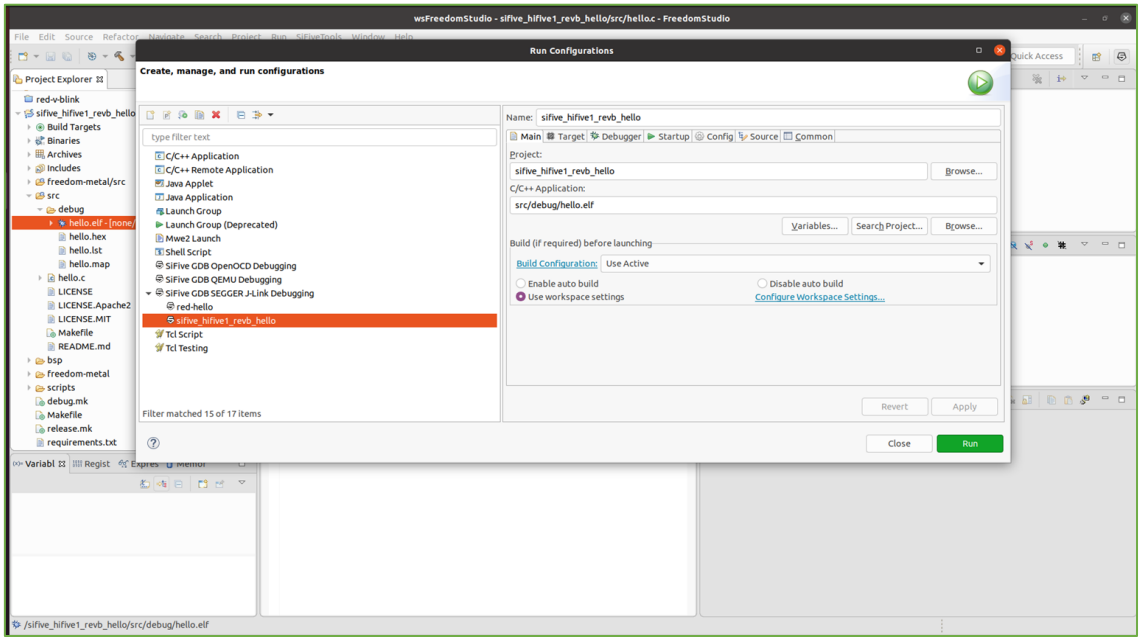

Figura 5-14. Compilación.
Fuente: Elaboración propia.

Se puede ver el resultado de la ejecución por consola, tal y como se muestra en la Figura 5-15 en *console* (zona central superior).

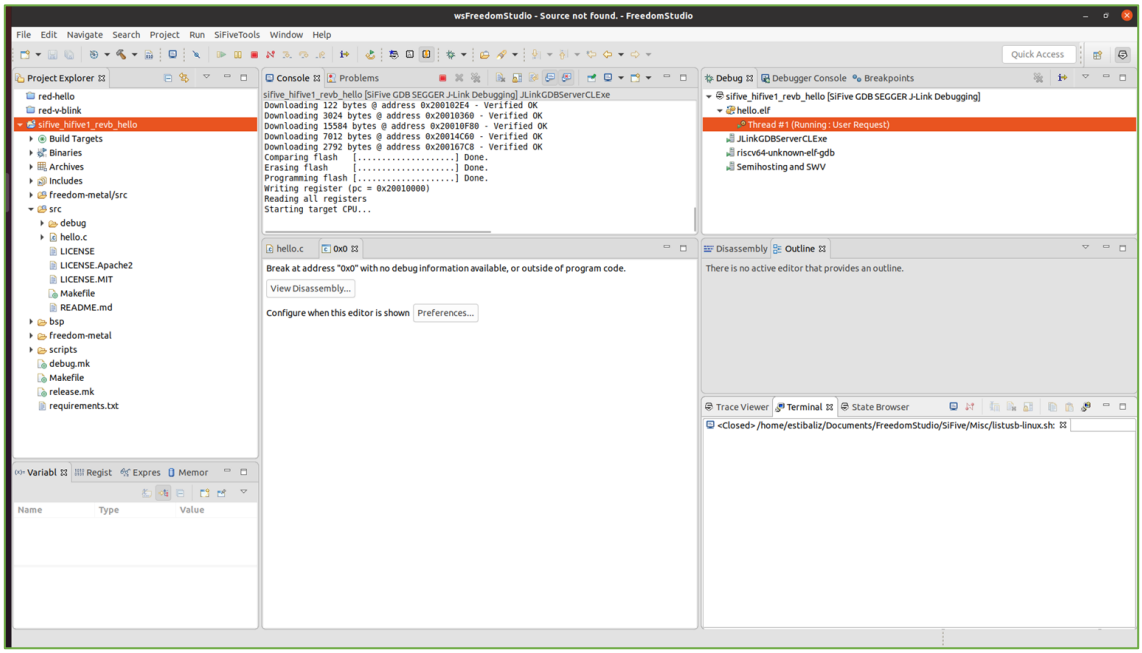

Figura 5-15. Configuración de la compilación.
Fuente: Elaboración propia.

Para mostrar el resultado de la ejecución del código (“*Hello, Word!”)* por la terminal, se debe seleccionar el puerto de la placa, en *serial port -> /dev/ttyACM1* (Figura 5-16); posteriormente pulsamos en *Ok* y así, aparecerá el resultado en la terminal, tal y como aparece en la Figura 5-17 en la parte inferior en el cuadro rojo.

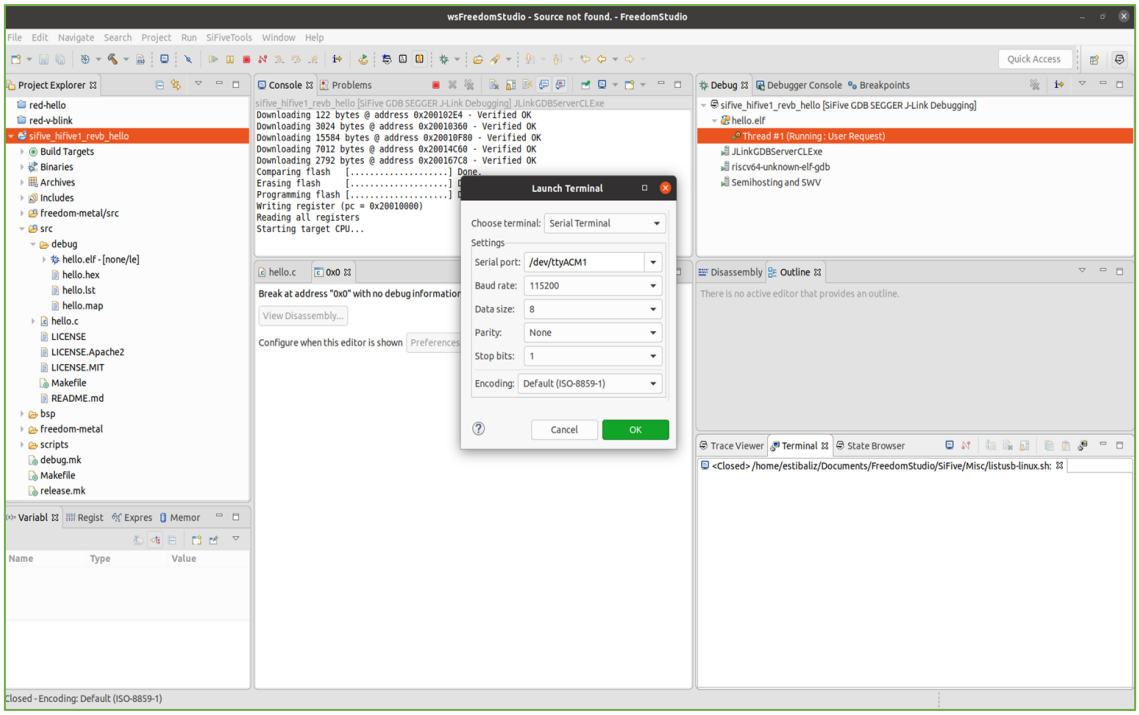

Figura 5-16. Resultado de compilación Hello World.
Fuente: Elaboración propia.

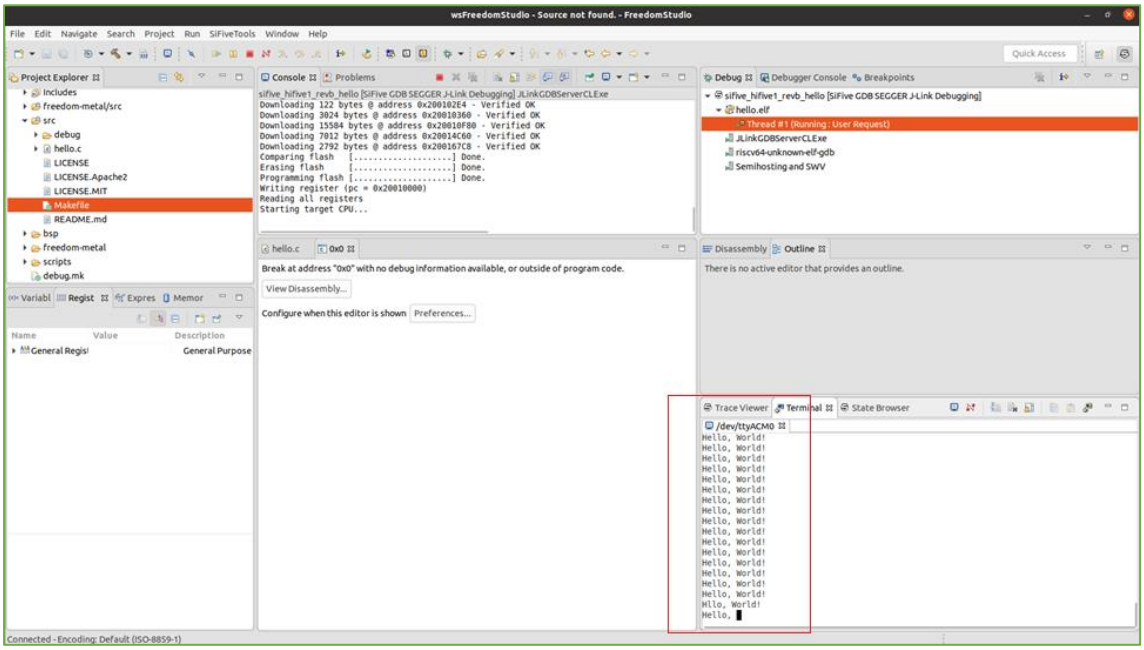

Figura 5-17. Configuración de la placa.
Fuente: Elaboración propia.

Ahora se creará otro ejemplo; para ello, se borra el código y se toma el mismo ejemplo realizado en PlatformIO, para sumar cada componente de un vector. Se escribe el código en C y se guarda *(ctrl+s)*.

A continuación, Figura 5-18, se pulsará en *run->debugconfigurations* y se seleccionará el trabajo a ejecutar. El programa selecciona automáticamente la primera instrucción del código en C y, después, se pulsa en el botón *Play* (cuadrado rojo) para ejecutar el código. En la parte de la derecha aparece el código en ensamblador.

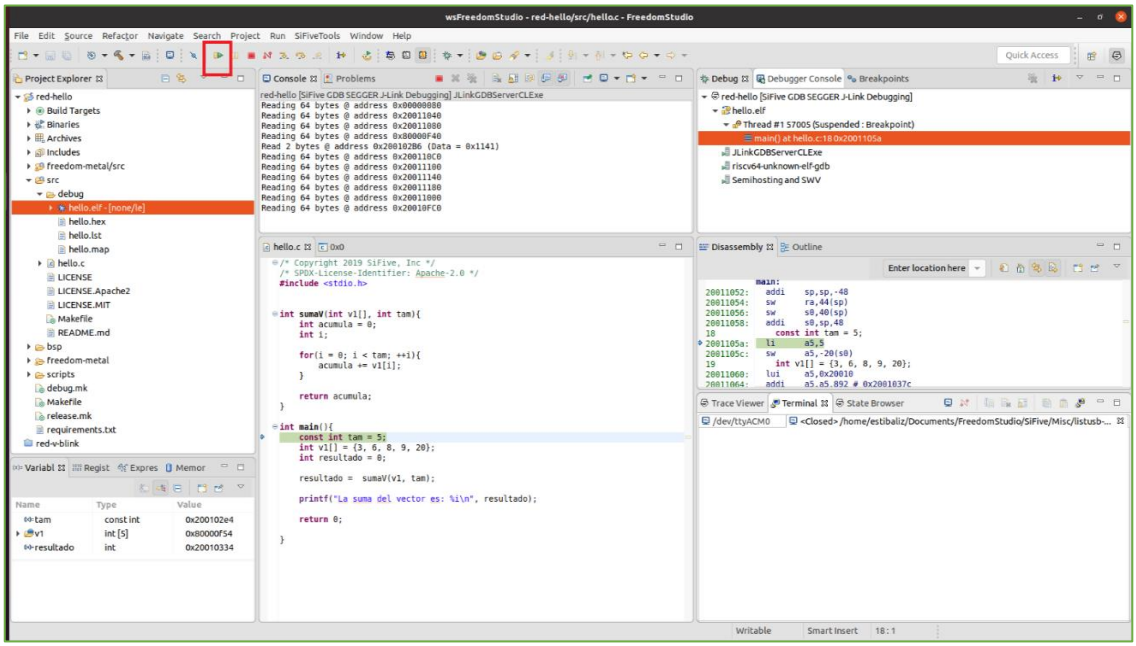

Figura 5-18. Depuración de la suma de los elementos de un vector en C.
Fuente: Elaboración propia.

Cuando el programa termina, se muestra el resultado del código por la terminal (cuadrado rojo) en la Figura 5-19, eligiendo la pestaña */dev/ttyACM0* en la consola inferior derecha. Además, aparecerá el resultado de la suma de las componentes del vector en la zona señalada en morado (Figura 5-19).

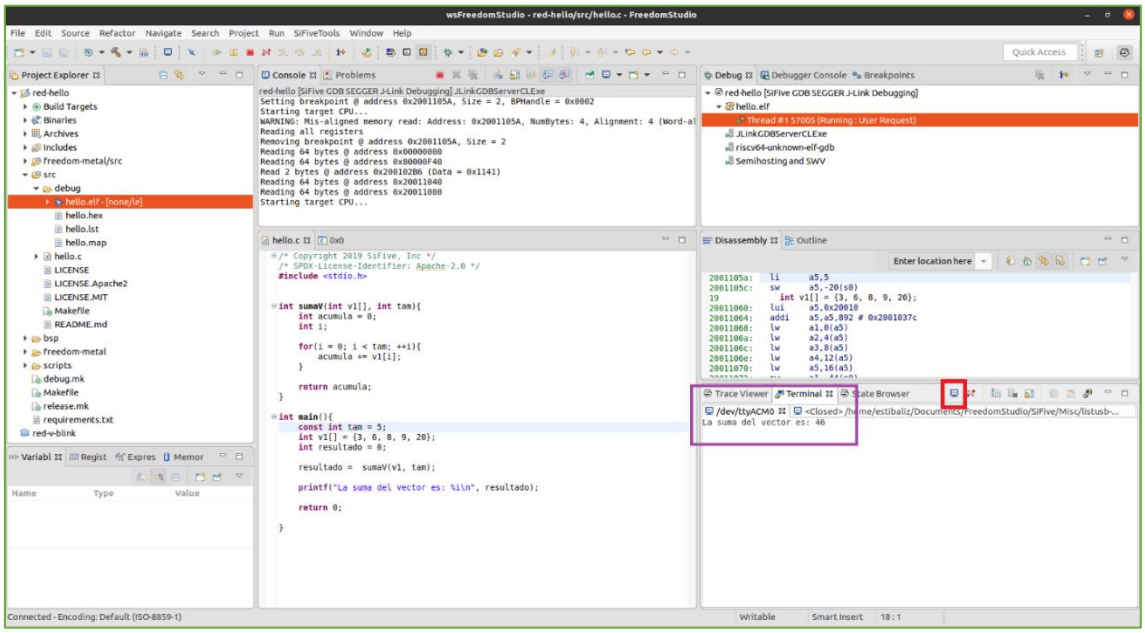

Figura 5-19. Resultado de ejecución de la suma de los elementos de un vector.
Fuente: Elaboración propia.

Con los apartados 5.1 y 5.2 se da como finalizado el Tutorial para desarrollo de programas en C, tanto con Visual Studio Code como con Freedom Studio con la placa SparkFun Red-V Thing Plus.

Este IDE resulta ser muy completo y la interfaz es mejor que la del Visual Studio Code. Además, se parece al programa Eclipse utilizado en la asignatura de Estructura de Computadores del Grado de Ingeniería Informática por lo que su utilización resulta muy intuitiva, aunque las funcionalidades respecto al VSC son muy parecidas. También existe mucha información en Internet donde aparecen ejemplos para entender el funcionamiento de este IDE.

A continuación, en los apartados 5.3 y 5.4, se explicarán unos tutoriales para desarrollo de programas en lenguaje ensamblador utilizando estos mismos entornos de desarrollo.

5.3 Tutorial para desarrollo de programas en ensamblador usando Visual Studio Code

Para escribir lenguaje ensamblador en Visual Studio Code se deben seguir los siguientes pasos:

1. Instalar la extensión *RISC-V Venus Simulator.* Para ello, hay que buscar la extensión poniendo *"riscv"* en el buscador de extensiones y cliqueamos en el botón de *"install",* Figura 5-20.

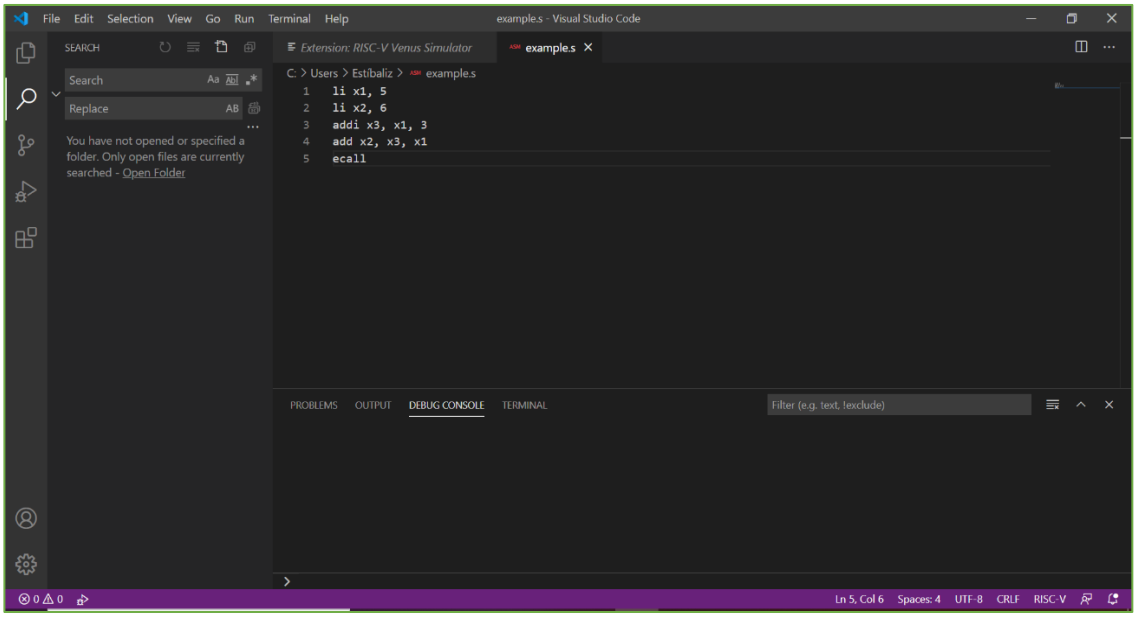

Figura 5-20. Código ensamblador RISC-V.
Fuente: Elaboración propia.

2. Una vez instalada la extensión, hay que pulsar en el botón *File-> new file* y se creará un archivo con extensión ".s"; en dicho archivo se escribirá el código a ejecutar como en la Figura 5-21. En este caso de ejemplo consiste en realizar una suma dando como resultado 13 (0xD en hexadecimal) que se almacenará en el registro x2.

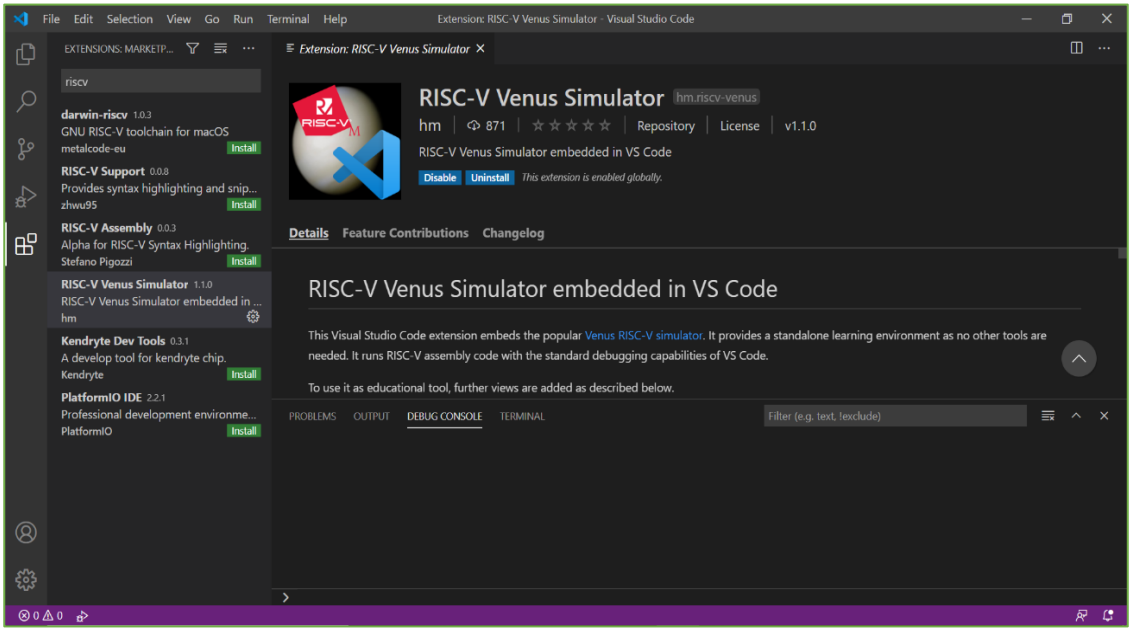

Figura 5-21. Instalación RISC-V Venus Simulator.
Fuente: Elaboración propia.

3. Una vez escrito y guardado el código, se procederá a realizar la compilación y la depuración con el botón *"Run and Debug"* como se muestra en la Figura 5-22.

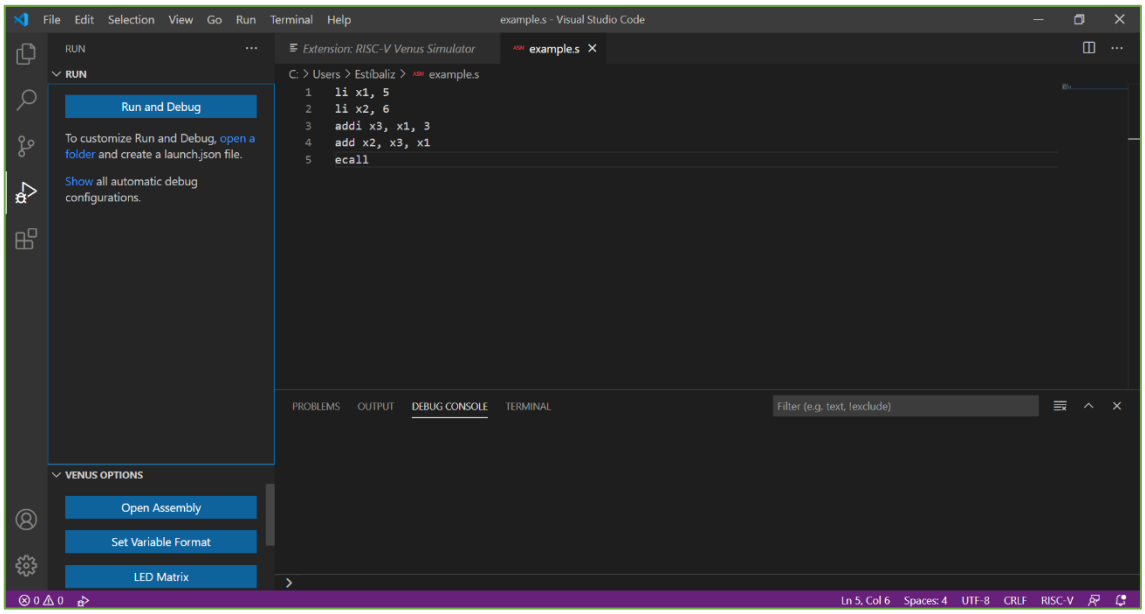

Figura 5-22. Run and Debug.
Fuente: Elaboración propia.

Al pulsar el botón *"Run and Debug"* se verá cómo van cambiando los registros paso por paso como indica la Figura 5-23.

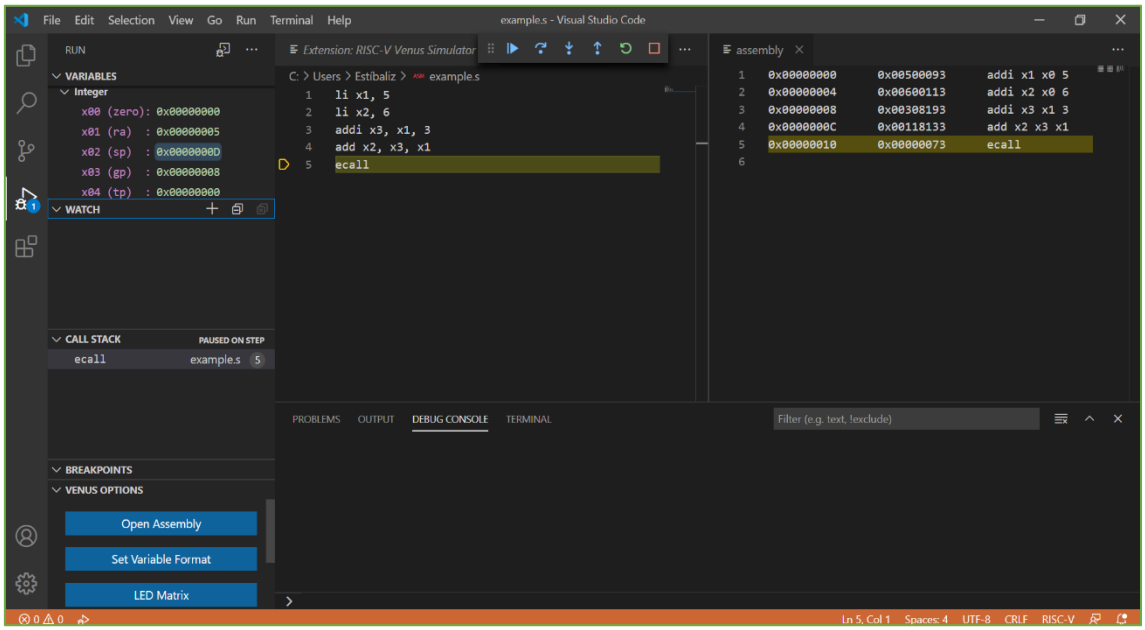

Figura 5-23. Interfaz de Run and Debug.
Fuente: Elaboración propia.

Finalmente, en la Figura 5-24 se muestra cómo cambiaría el contenido de la memoria si hubiese instrucciones load o store.

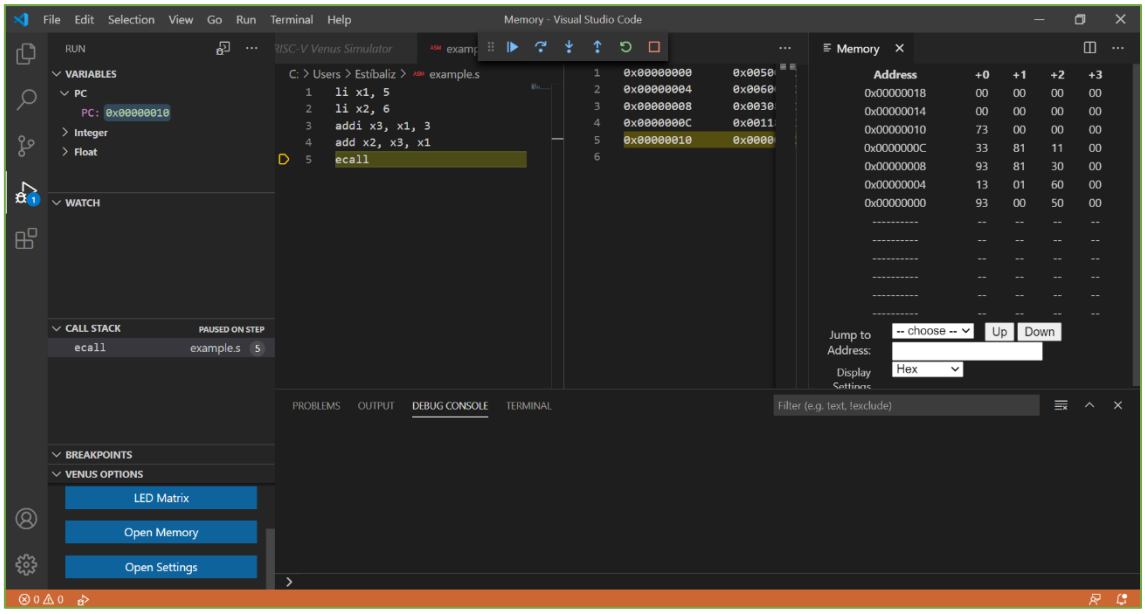

Figura 5-24. Interfaz Run and Debug.
Fuente: Elaboración propia.

Así es como se realiza un programa en ensamblador con código RISC-V utilizando la herramienta Visual Studio Code. Gracias a la extensión, RISC-V Venus Simulator, instalada resulta sencillo escribir códigos en ensamblador. Además, se puede ver el contenido de la memoria y cómo cambian los registros. La interfaz resulta sencilla así como su utilidad.

5.4 Tutorial para el desarrollo de programas en ensamblador con Freedom Studio

En esta ocasión, se utilizará el emulador Qemu [27] dentro del propio Freedom Studio. Para ello, se deberán seguir los siguientes pasos:

1. Seleccionar un nuevo proyecto en la barra del menú en el botón de la *"S"* (cerca del botón *Debug).*
2. Cuando aparezca una pantalla como la Figura 5-25, seleccionar en *Target "qemusifive-e31".*
3. Escribir un nombre al nuevo proyecto.
4. Pulsar el botón *Finish.*

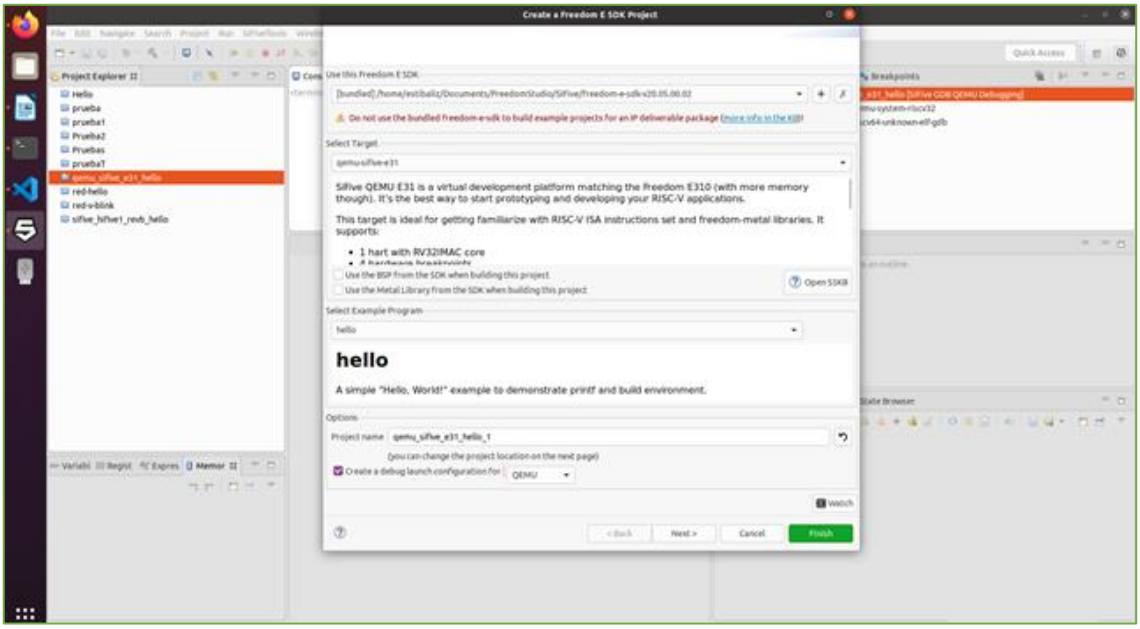

Figura 5-25. Interfaz creación del proyecto.
Fuente: Elaboración propia.

Una vez creado el proyecto, aparecen diferentes archivos por defecto, Figura 5-26. En el nombre del proyecto, se pulsará el botón derecho del ratón *"New->file"*. Es ahí donde se crearán los archivos: .S y .c. Así que, primero hay que crear el archivo .S (es importante que la "s" sea mayúscula) que se llamará en este caso de ejemplo "operaciones.S" y después, el archivo .c que se denominará "hello.c" donde se escribirá el código en C. Se ejecutará primero el código "hello.c" que llamará a "operaciones.S" donde se encuentran las operaciones en ensamblador. Este es un ejemplo que mezcla C con ensamblador.

.

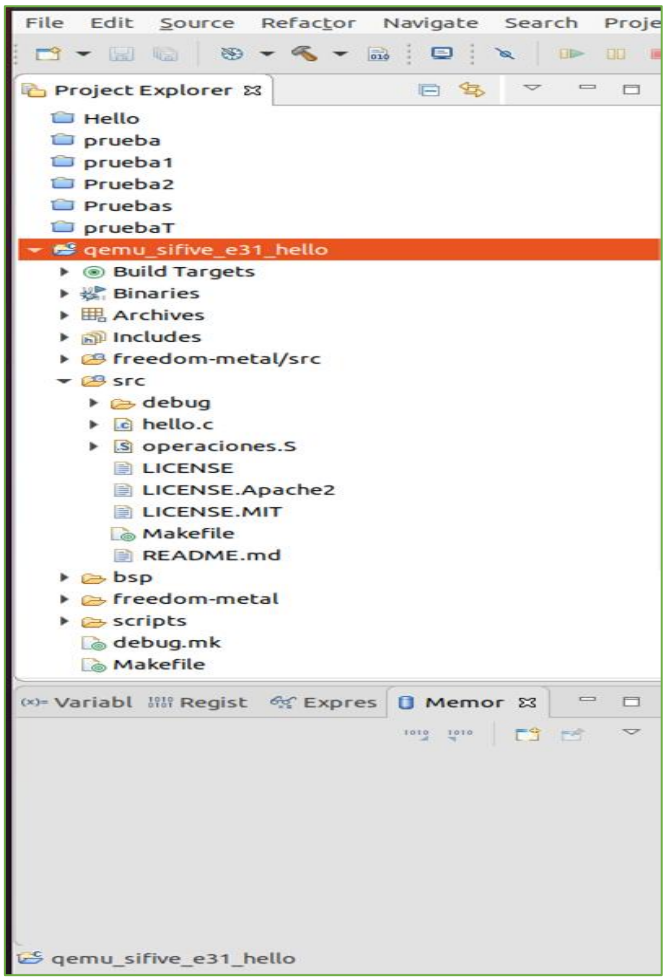

Figura 5-26. Explorador de proyectos.
Fuente: Elaboración propia.

En el archivo *operaciones.S*, se escribirá el código en ensamblador (Figura 5-27); en este caso, la rutina f1 realiza una suma y f2 se ejecuta después de haberse ejecutado f1, que es una resta, cuyo resultado final se mostrará en la Figura 5-31.

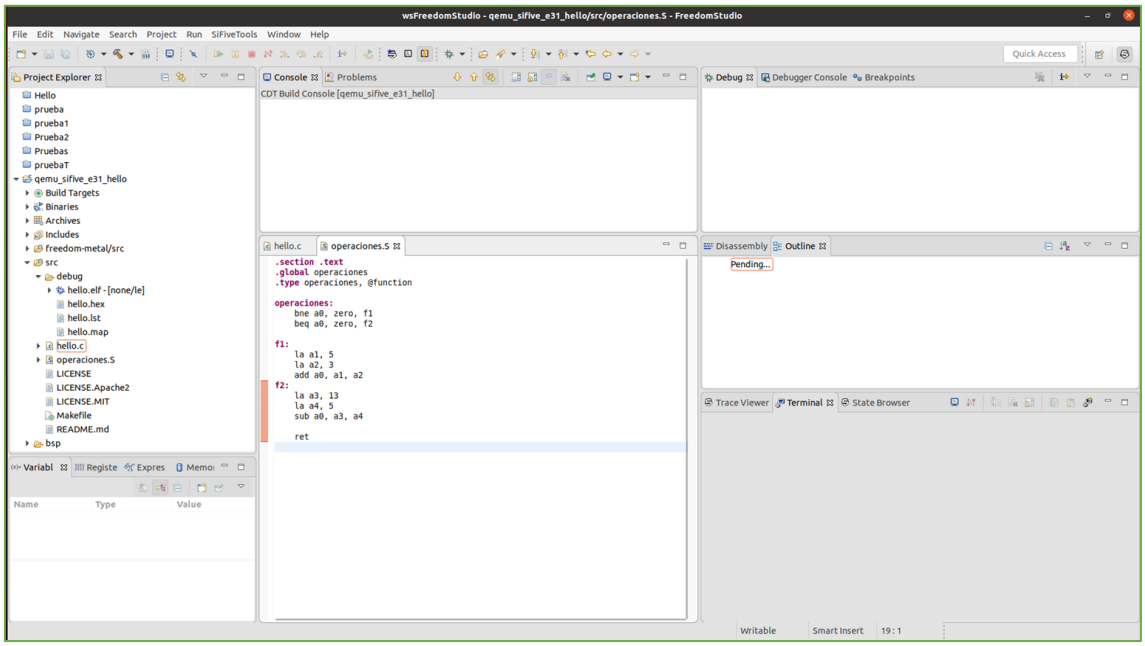

Figura 5-28. Código en ensamblador. Archivo operaciones.S.
Fuente: Elaboración propia.

En el archivo *hello.c* se escribirá el código en C (Figura 5-28), para mostrar por pantalla el resultado de las operaciones anteriores.

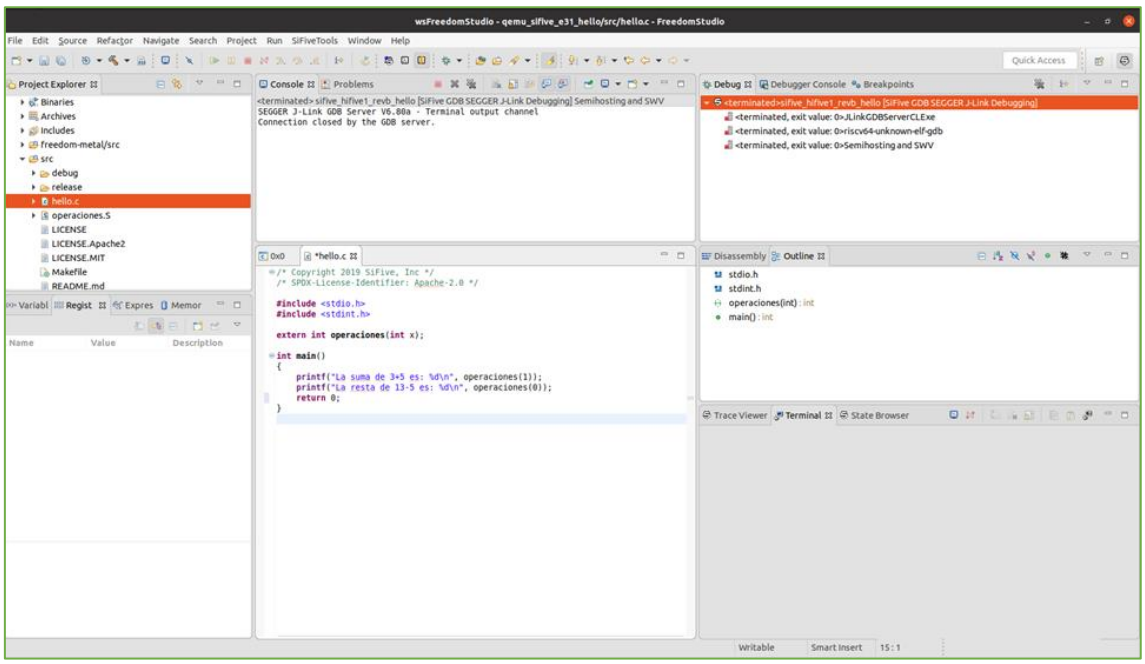

Figura 5-27. Código en C. Archivo hello.c
Fuente: Elaboración propia.

Una vez creados estos dos archivos y con el código introducido, se tecleará *"ctrl+s"* para guardar el proyecto y se pulsará el botón del Martillo, el cual construirá el proyecto.

Después, se pulsará el botón de Debug (Figura 5-29).

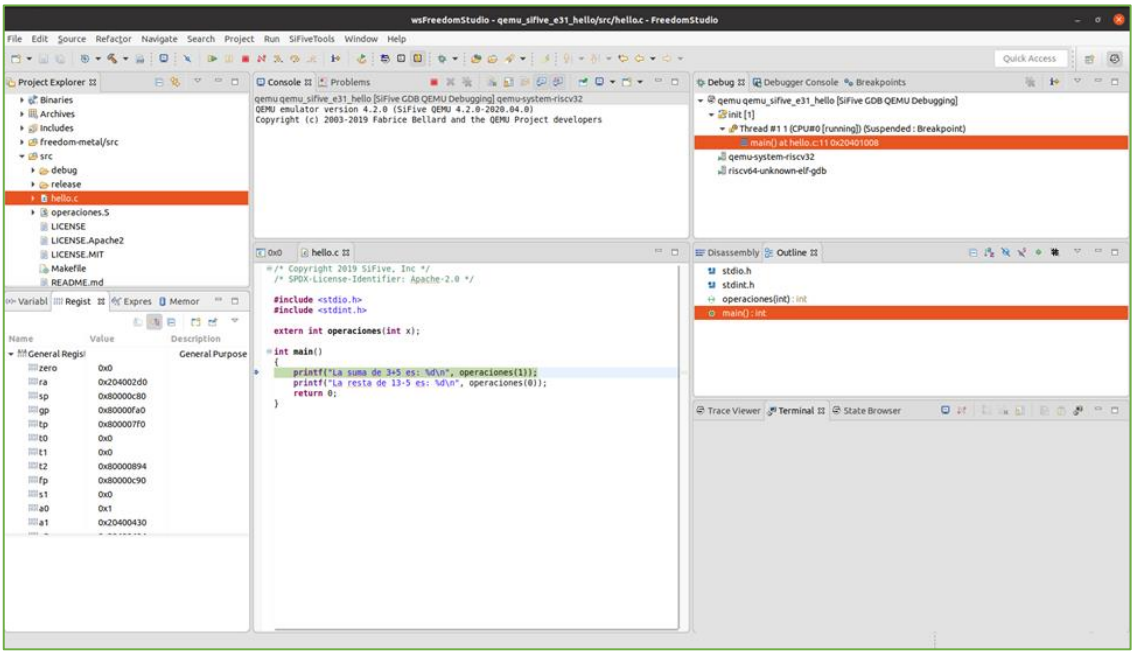

Figura 5-29. Depuración del código.
Fuente: Elaboración propia.

Se puede ejecutar el código paso por paso con la flechita situada al lado del botón *Play.* Además, se puede ver el contenido de los registros en la esquina inferior izquierda, tal y como se muestra en la Figura 5-30.

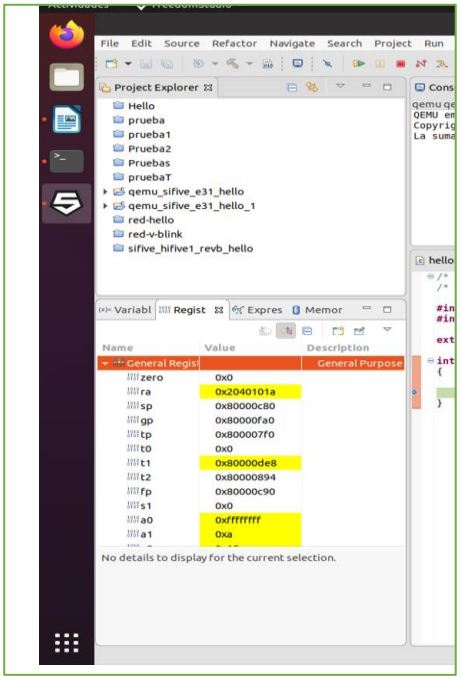

Figura 5-30. Visualización de los registros.
Fuente: Elaboración propia.

A continuación, se cliqueará en el botón del Play, el cual mostrará por pantalla el resultado de las operaciones tal y como aparece en la Consola (Figura 5-31).

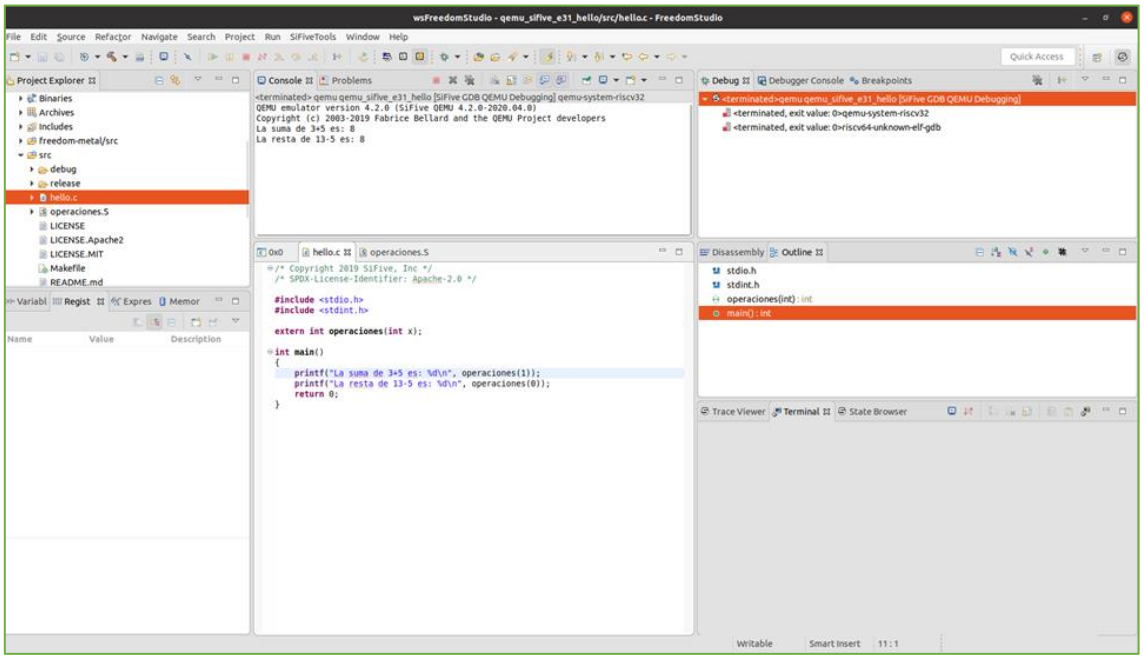

Figura 5-31. Resultado de la compilación.
Fuente: Elaboración propia.

Así es, como se realizan programas con código ensamblador en Freedom Studio.

Como conclusión cabe destacar que realizar programas con código ensamblador en Freedom Studio resulta simple debido a que como se ha dicho anteriormente la interfaz resulta sencilla de entender gracias a las asignaturas de computadores estudiadas durante el grado. Este IDE permite depurar y ver cómo van cambiando el contenido de los registros y realizar otras muchas más opciones.

Por último, para finalizar el presente capítulo resaltar que gracias a las asignaturas estudiadas durante el Grado de Ingeniería Informática, los IDEs utilizados en este Trabajo de Fin de Grado resultan fáciles de entender. Permiten depurar, compilar y volcar en la placa.

Además, existe mucha información en Internet y se van realizando mejoras en las actualizaciones. El IDE Freedom Studio es mejor por la interfaz que tiene que como se ha nombrado antes se parece a la de Eclipse, lo cual facilita su comprensión y su utilización a la hora de realizar códigos.

Capítulo 6 - Conclusiones

Una vez finalizado el proceso de elaboración del presente TFG con la documentación y herramientas utilizadas se procederá a señalar las conclusiones más importantes de dicho proyecto.

En primer lugar, se ha conseguido el primero de los objetivos que era el estudio de la arquitectura RISC-V y su ecosistema Hardware y Software. Sin embargo, no se han podido utilizar las placas Sipeed Longan Nano y Arty A7, tal y como se pretendía durante los primeros momentos debido a dificultades (mencionadas en el capítulo 1) provocadas por:

a) El confinamiento provocado por la emergencia sanitaria del COVID-19 tuvo una fuerte repercusión a la hora de no poder contar con los medios presenciales, ni materiales necesarios desde mi domicilio donde tuve que pasar dicho confinamiento.

b) Los reiterados intentos para el funcionamiento de las placas no tuvieron el éxito deseado, tal y como se hubiera esperado:

    - Se hicieron particiones en el equipo instalando Ubuntu por falta de espacio en el disco duro, llegando a utilizar otros equipos para comprobar su funcionamiento.
    - Se realizaron pruebas con un voltímetro para medir la corriente en los distintos puntos de la placa para comprobar si estaban o no defectuosas.
    - Falta de documentación en español, la mayoría en inglés y en chino, así como la falta de respuestas en los foros destinados a este propósito.

No obstante, tal y como se ha comenzado este capítulo, se ha conseguido el principal objetivo sobre el estudio de la arquitectura RISC-V. Aunque la información ha sido demasiado extensa, el hecho de haber cursado asignaturas relacionadas con computadores durante el Grado de Informática, tales como Fundamentos de Computadores II - estudiando la arquitectura ARM - y, en particular, Arquitectura de Computadores, ha facilitado el seguimiento de dicha información para su mejor comprensión.

Durante el Grado, también se han utilizado distintos simuladores o IDES como el Eclipse. Esto ha facilitado la tarea del estudio del ecosistema Software, ya que mediante este trabajo hemos aprendido el funcionamiento de distintos simuladores de esta

arquitectura. Con ello ha resultado ser Ripes el simulador más completo debido a todas las funcionalidades que proporciona, tal y como se ha mostrado en el Capítulo 3.

En lo que se refiere a su ecosistema Hardware, hemos conocido distintas placas que pueden utilizarse para probar esta arquitectura y numerosas empresas que tratan con esta arquitectura. Aunque durante el Grado también se usaron algunas, eran diferentes por el tipo de arquitectura que empleaban. Pese a los mencionados problemas con algunas de ellas, con esfuerzo e interés por sacar este trabajo adelante, se ha podido conseguir el segundo objetivo, procediendo a analizar y estudiar la placa *SparkFun Red-V Thing Plus*, que no se había utilizado durante el Grado. Para ello, se han preparado unos tutoriales básicos de cómo hacer programas en C y en ensamblador, usando dos IDE distintos, PlatformIO en Visual Studio Code y Freedom Studio de SiFive, realizando capturas de pantalla de todos los pasos para que resulte más sencillo utilizarlos. Con la realización de estos tutoriales hemos aprendido a usar este tipo de IDEs, placas y lenguaje ensamblador.

Por otro lado, si nos parásemos a reflexionar sobre posibles líneas futuras de actualización basadas en este TFG, mencionaré algunas de ellas que se pueden tener en cuenta:

a) Sería interesante aprovechar las dificultades con otras placas como la Sipeed Longan Nano y la Arty A7 e intentar averiguar las causas de dichos problemas encontrados para que en sucesivas ocasiones no se caiga en el mismo error o no se trabaje durante tantas horas en vano.

b) Una vez analizadas las causas, se podría crear un tutorial de cómo funcionan o compartir lo averiguado para poder avanzar, ya que así es como funciona el proyecto RISC-V compartiendo los conocimientos y avanzar en la investigación.

c) Por último, considero que sería de gran provecho la utilización de esta arquitectura en asignaturas de computadores, puesto que actualmente se utiliza la arquitectura ARM, y RISC-V podría convertirse en la arquitectura del futuro. Así, se ganaría independencia tecnológica, reduciendo los riesgos de monopolio y facilitando el desarrollo de nuevas aplicaciones. Además, el poder utilizar placas tan pequeñas permitiría llevarlas cómodamente a casa y no solo poder trabajar con ellas en los laboratorios de la Facultad.

Por todo ello, damos por alcanzados todos los objetivos y concluimos el trabajo realizado, mencionando un futuro trabajo como la extensión del mismo y la utilización de otras placas solventando los problemas originados con las mismas.

Chapter 6 – Conclusions

Once the process of preparing this FDP with the documentation and tools used is finished, the most important conclusions of said project will be pointed out.

At the beggining, the first of the objectives has been got, which was the study of the RISC-V architecture and its Hardware and Software ecosystem. However, the Sipeed Longan Nano and Arty A7 plates could not be used, as intended during the first moments due to difficulties (mentioned in chapter 1) caused by:

a) The confinement caused by the health emergency of COVID-19 had a strong impact on not being able to have the face-to-face means or the necessary materials from my home where I had to pass it.
b) The repeated attempts to operate the boards did not have the desired success, as would have been expected:

    - Partitions were made on the computer by installing Ubuntu due to lack of space on the hard disk, even using other computers to check its operation.
    - Tests were carried out with a voltmeter to measure the current in the different points of the board to check if they were defective or not.
    - Shortage of documentation in Spanish, the majority in Chinese, as well as the shortage of responses in the forums for this purpose.

However, as this chapter has begun, the main objective of the study of the RISC-V architecture has been got. Although the information has been too extensive, the fact of having taken subjects related to computers during the Computer Degree, such as Computer Fundamentals II - studying ARM architecture - and, in particular, Computer Architecture, has facilitated the monitoring of said information for your better understanding.

During the Degree, different simulators or IDEs such as the Eclipse have also been used. It has facilitated the task of studying the Software ecosystem, since through this work we have learned the operation of different simulators of this architecture. With this it has turned out to be Ripe, the most complete simulator due to all the functionalities it provides, as shown in Chapter 3.

When it comes to your Hardware ecosystem, we have seen different boards that can be used to test this architecture and many companies dealing with this architecture. Although some were also used during the Degree, they were different due to the type of architecture they used. Despite the aforementioned problems with some of them, with effort and interest to carry out this work, the second objective has been achieved,

proceeding to analyze and study the SparkFun Red-V Thing Plus board, which had not been used during the Degree. For this, some basic tutorials have been prepared on how to make C and assembler programs, using two different IDE's, PlatformIO in Visual Studio Code and Freedom Studio from SiFive, taking screenshots of all the steps to make them easier to use. With the completion of these tutorials, we have learned to use this type of IDE´s boards and assembly language.

Furthermore, if we stop to reflect on possible future update lines based on this FDP, I will mention some of them that can be taken into account:

a) It would be interesting to take advantage of the difficulties with other plates boards such as the Sipeed Longan Nano and the Arty A7 and try to find out the causes of these problems found so that on successive occasions the same error is not made or it is not worked during so many hours in vain.
b) Once the causes have been analyzed, a tutorial could be created on how they work or share what was found in order to move forward, this is how the RISC-V project works, sharing knowledge and advancing the investigation.
c) Finally, I consider that the use of this architecture in computer subjects would be of great benefit, since the ARM architecture is currently used, and RISC-V could become the architecture of the future. So, technological independence would be gained, reducing monopoly risks and facilitating the development of new applications. In addition, being able to use such small boards would allow them to be carried comfortably home and not only to be able to work with them in the laboratories of the Faculty.

For all this, we consider all the objectives achieved and we conclude this project, mentioning future work such as its extension and the use of other boards, solving the problems caused by them.

Bibliografía

Todas las direcciones URL que aparecen a continuación, han sido comprobadas y validadas a fecha de **20 de noviembre de 2020.**

[1] *Origen RISC-V. Investigación en Berkeley*. (2020) Obtenido de *https://riscv.org/about/history/*

[2] Castañeda, A. (2 de septiembre de 2019). *Qué es RISC-V, el hardware abierto sin límites.* Obtenido de *https://www.zonamovilidad.es/que-es-risc-v-hardware-abierto-sin-limites*

[3] Ortiz, J. J. (20 de septiembre de 2018). *Grado en Ingeniería del Software*. Obtenido de *http://www.fdi.ucm.es/profesor/jjruz/EC-IS/Temas%02-Arquitectura%20del%20procesador.pdf*

[4] (13 de junio de 2019). *Architecnologia.* Obtenido de *https://architecnologia.es/risc-v-introduccion-a-la-isa-parte-2*

[5] Lanchares Dávila, J. (S. d.). *Apuntes de Estructura de Computadores.* Departamento de Arquitectura de Computadores y Automática (UCM). Obtenido de *http://www.dacya.ucm.es/lanchares/documentos/2.9.5%20Apuntes%20de%20Estructura%20de%20Computadores.pdf*

[6] Patterson, D. & Waterman, A. (11 de julio de 2018). *Guía práctica de RISC-V: El Atlas de una Arquitectura Abierta.* Strawberry Canyon LLC. (Primera Edición, 1.0.5). Obtenido de *http://riscvbook.com/spanish/guia-practica-de-risc-v-1.0.5.pdf*

[7] Waterman, A., Asanović**,** K., SiFive Inc., CS Division, EECS Department, University of California, Berkeley (December 13, 2019). *The RISC-V Instruction Set Manual. Volume I: Unprivileged ISA*. Version 20191213. Obtenido de *https://riscv.org//wp-content/uploads/2019/12/riscv-spec-20191213.pdf*

[8] Williams, D. (11 de agosto de 2014). *Arquitectura del Computador*. Obtenido de *https://deywilliamsqs.wordpress.com/2014/08/11/formato-general-de-las-instrucciones/*

[9] Hennessy, J. & Patterson, D. (2019). *Computer Architecture: A Quantitative Approach.* (Sixth Edition). Morgan Kaufmann.

[10] Wikipedia. (4 de mayo de 2020). Obtenido de *modos de direccionamiento https://es.wikipedia.org/wiki/Modo_de_direccionamiento*

[11] *Estructura de máquinas - Introducción a RISC-V*. (S.d.). Obtenido de *https://cc-3.github.io/notes/02_Intro-RISCV/*

[12] Wikipedia. (2 de noviembre de 2020). *Unidad Central de Procesamiento*. Obtenido de *https://es.wikipedia.org/wiki/Unidad_central_de_procesamiento*

[13] Patterson, D. & Hennessy, J. (Julio, 2004). *Estructura y Diseño de Computadores*. Interficie circuitería / Programación. 3 Apéndices Edit. Reverté, S.A.

[14] Universidad Autónoma de Barcelona (UAB). Sala de Prensa. *Proyectos recientes destacados.* (4 de febrero de 2020). Obtenido de *https://www.uab.cat/web/sala-de-prensa/detalle-noticia/se-crea-la-red-riscv-para-impulsar-el-desarrollo-de-hardware-de-codigo-abierto-1345667994339.html?noticiaid=1345806981489*

[15] Pueyo Busquets, J. (10 de junio de 2019). *El nuevo superordenador*. Obtenido de *https://elpais.com/tecnologia/2019/06/10/actualidad/1560150491_493847.html*

[16] RISC-V. (6 de agosto de 2020). *Available Software.* Obtenido de *https://riscv.org/exchange/software*

[17] SiFive. (2020). *SiFive Software*. Obtenido de *https://www.sifive.com/software*

[18] Andes. (2020). *RISC-V @ Andes*. Obtenido de *http://www.andestech.com/en/risc-v-andes/*

[19] SiFive. (S.d.). *SiFive Core IP*. Obtenido de *https://www.sifive.com/risc-v-core-ip*

[20] Digital, W. (2020). *RISC-V: Accelerating Next Generation Compute Requirements*. Obtenido de *https://www.westerndigital.com/company/innovations/risc-v*

[21] mgielda. (2020). *RISC-V Cores and SoC Overview*. Obtenido de *https://github.com/riscv/riscv-cores-list#soc-platforms*

[22] SiFive. (19 de agosto de 2019). *HiFive1*. Obtenido de *https://www.sifive.com/boards/hifive1*

[23] SiFive. (S. d.). *HiFive1 Rev B*. Obtenido de *https://www.sifive.com/boards/hifive1-rev-b*

[24] SiFive. (S.d). *HiFive Unleashed*. Obtenido de *https://www.sifive.com/boards/hifive-unleashed*

[25] SiFive. (2019). *SiFive FE310-G002 Manual v19p05*. Obtenido de *https://sifive.cdn.prismic.io/sifive%2F59a1f74e-d918-41c5-b837-3fe01ba7eaa1_fe310-g002-manual-v19p05.pdf*

[26] SparkFun Electronics. (2020). *RED-V Thing Plus Hookup Guide*. Obtenido de *https://learn.sparkfun.com/tutorials/red-v-thing-plus-hookup-guide*

[27] Qemu. (2020). Obtenido de *https://www.qemu.org*

Apéndices

Apéndice A: Otras nomenclaturas de RISC-V

Extensiones estándar

Debido al crecimiento en el número de extensiones, la Fundación RISC-V ha ideado otras nomenclaturas [7] que aparecen a continuación:

- **Nombres de extensión con números**

El nombre tiene tres partes: la especificación base utilizada (RV32I, RV64I, etc.), las extensiones estándar agregadas (M, F, A, etc.) y los números de versión de mayor a menor y separados por una “p”. (1p0 para la versión 1.0, etc.). En muchos casos, los números de versión se quitan para simplificar (RV32IC, RV64G, etc.). Por ejemplo, El estándar ISA de 64 bits original definido en la versión 1.0 se puede escribir en su totalidad como “RV64I1p0M1p0A1p0F1p0D1p0”, más simple “RV64I1M1A1F1D1”.

Si tuviéramos en cuenta este esquema con la segunda versión, "RV32I" sería “RV32I2”.

- **Guion bajo**

Los guiones bajos "_" se pueden utilizar para separar las extensiones del ISA evitando errores. Debido a que la extensión "P" para SIMD empaquetada puede confundirse con el punto decimal en un número de versión, debe ir precedida de un guion bajo si sigue a un número. Por ejemplo, "RV32I2p2" significa la versión 2.2 de RV32I, mientras que "RV32I2_p2" significa la versión 2.0 de RV32I con la versión 2.0 de la extensión P.

- **Nombres de extensión estándar adicionales**

Con el crecimiento en el número de extensiones, las extensiones estándar también se pueden nombrar usando una sola "Z" seguida de un nombre alfabético y un número de versión opcional. Por ejemplo, "Zifencei" nombra la extensión y “Zifencei2” y “Zifencei2p0” nombran la versión 2.0 del mismo.

A diferencia de las extensiones de un solo carácter, las extensiones Z deben estar separadas por guiones bajos, agrupadas por categoría y, después, alfabéticamente dentro de cada categoría. Por ejemplo, Zicsr_Zifencei_Zam.

Extensiones no estándar

A diferencia de las extensiones estándar diseñadas para no entrar en conflicto con ninguna otra estándar; la **extensión no estándar** puede entrar en conflicto con otros estándares. Se nombran con una sola "X" seguida de un nombre alfabético y un número de versión opcional. Por ejemplo, "Xhwacha" nombra la extensión ISA Hwacha; “Xhwacha2” y “Xhwacha2p0” nombran la versión 2.0 del mismo.

Además, las extensiones no estándar deben aparecer después de todas las extensiones estándar. Si se enumeran varias extensiones no estándar deben estar separadas por un guión bajo y ordenadas alfabéticamente; por ejemplo, un ISA con extensiones no estándar Argle y Bargle: "RV64Izifencei_Xargle_Xbargle".

Apéndice B: Modos de la CPU

Como ya se mencionó en el capítulo 2, RISC-V posee otros dos nuevos modos más privilegiados que el modo Usuario: modo Máquina que ejecuta el código más fiable y el modo Supervisor que provee soporte para sistemas operativos como Linux, FreeBSD y Windows. La información de este apéndice es un resumen obtenido principalmente de *Patterson, D. y & Waterman, A. (2018). Guía práctica de RISC-V: El Atlas de una Arquitectura Abierta* y *Waterman, A.*

Modo Máquina

En RISC-V existen ocho registros de control y estado (CSRs) para el manejo de excepciones en modo máquina, los cuales se muestran a continuación [6]:

- **mstatus,** Estado de Máquina, contiene el habilitador global de interrupciones, junto con otros estados, como muestra la Figura 0-1. Los únicos campos presentes en procesadores simples, sólo con modo máquina y sin las extensiones F ni V, son el habilitador global de interrupciones MIE y MPIE, el cual después de una excepción contiene el valor anterior de MIE.

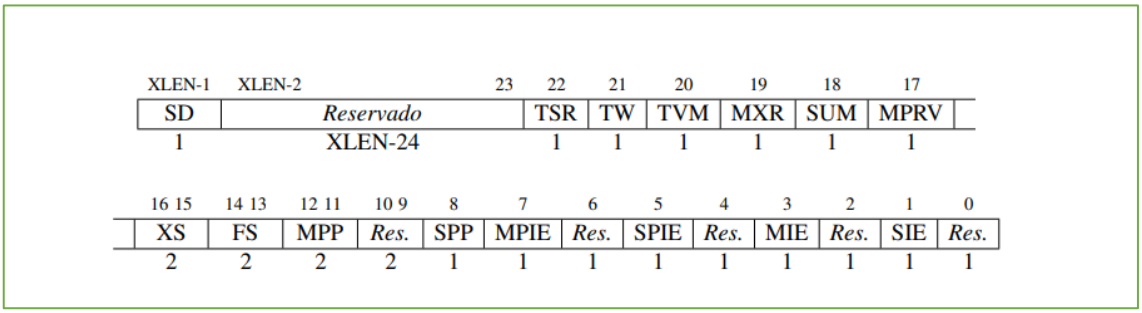

Figura 0-1. CSR mstatus.
XLEN es 32 para RV32 o 64 para RV64.
Fuente: Patterson, D. & Waterman, A. (11 de julio de 2018). Guía práctica de RISC-V: El Atlas de una Arquitectura Abierta. Strawberry Canyon LLC. (Primera Edición, 1.0.5).

- **mip**, Interrupciones de Máquina Pendientes, lista las interrupciones actualmente pendientes. El CSR se muestra en la Figura 0-2.
- **mie**, Habilitador de Interrupciones de Máquina, lista que interrupciones puede tomar el procesador y cuáles no. El CSR se muestra en la Figura 0-2.

XLEN-1	12	11	10	9	8	7	6	5	4	3	2	1	0
Reservado		MEIP	*Res.*	SEIP	*Res.*	MTIP	*Res.*	STIP	*Res.*	MSIP	*Res.*	SSIP	*Res.*
Reservado		MEIE	*Res.*	SEIE	*Res.*	MTIE	*Res.*	STIE	*Res.*	MSIE	*Res.*	SSIE	*Res.*
XLEN-12		1	1	1	1	1	1	1	1	1	1	1	1

Figura 0-2. CSRs de interrupciones de máquina.
XLEN es 32 para RV32, o 64 para RV64.
Fuente: Patterson, D. & Waterman, A. (11 de julio de 2018). Guía práctica de RISC-V: El Atlas de una Arquitectura Abierta. Strawberry Canyon LLC. (Primera Edición, 1.0.5).

Los CSRs de interrupciones máquina son registros de XLEN-bits de lectura y escritura que contienen los bits de interrupciones pendientes (mip) y habilitadores de interrupciones (mie). Solo es posible escribir en los bits correspondientes a las interrupciones del menor privilegio (SSIP), interrupciones de temporizador (STIP) e interrupciones externas (SEIP) en mip por medio de esta dirección CSR, el resto de los bits son de solo lectura.

- **mcause,** Causa de Excepción de Máquina, indica qué excepción ocurrió (Figura 0-3). En ella se puede ver el CSR, en este se escribe un código que indica el evento que provocó la excepción (en el campo código de excepción). El bit de interrupción se pone a 1 si la excepción fue causada por una interrupción. La Figura 2-5 vista en el Capítulo 2 mapea los valores de los códigos indicando por que se ha producido esa excepción.

XLEN-1	XLEN-2	0
Interrupción	Código de Excepción	
1	XLEN-1	

Figura 0-3. CSRs de causa para máquina y supervisor (mcause y scause).
Fuente: Patterson, D. & Waterman, A. (11 de julio de 2018). Guía práctica de RISC-V: El Atlas de una Arquitectura Abierta. Strawberry Canyon LLC. (Primera Edición, 1.0.5).

- **mtvec**, Vector de Interrupciones de Máquina, contiene la dirección a la cual salta el procesador cuando ocurre una excepción (Figura 0-4). En ella se puede ver el

XLEN-1	2 1	0
BASE[XLEN-1:2]	MODE	
XLEN-2	2	

Figura 0-4. CSRs de direcciones base de vectores de excepciones para máquina y supervisor (mtvec y stvec).
Fuente: Patterson, D. & Waterman, A. (11 de julio de 2018). Guía práctica de RISC-V: El Atlas de una Arquitectura Abierta. Strawberry Canyon LLC. (Primera Edición, 1.0.5).

CSR, son registros de XLEN-bits de lectura y escritura que contienen la configuración de vectores de excepciones, que consiste en una dirección base del vector base (BASE) y un modo del vector (MODE). El valor del campo BASE siempre debe estar alineado en una frontera de 4 bytes. MODE = 0 significa que todas las excepciones escriben BASE en el PC. MODE = 1 escribe (BASE + (4 x causa)) en el PC en interrupciones asíncronas.

- **mtval,** Valor de Excepción de Máquina, contiene información adicional de excepciones: la dirección defectuosa para excepciones de direcciones, la instrucción misma para excepciones de instrucción ilegal, y cero para otras excepciones (Figura 0-5).
- **mepc,** PC de Excepción de Máquina, apunta a la instrucción donde ocurrió la excepción (Figura 0-5).
- **mscratch,** Scratch de Máquina, contiene una palabra de datos para almacenamiento temporal de manejadores de excepciones (Figura 0-5).

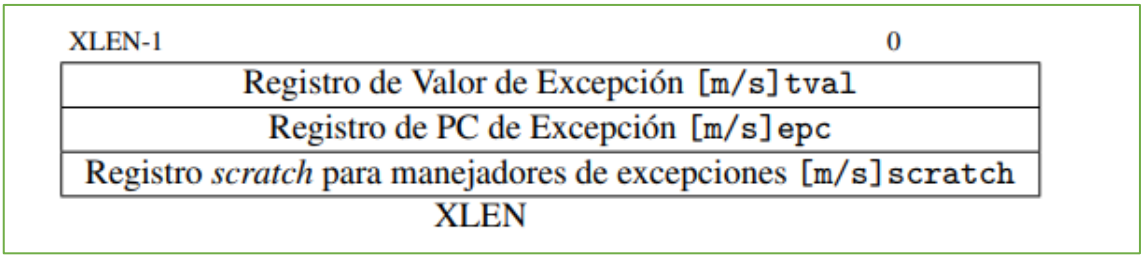

Figura 0-5. CSRs asociados con excepciones e interrupciones.
Fuente: Patterson, D. & Waterman, A. (11 de julio de 2018). Guía práctica de RISC-V: El Atlas de una Arquitectura Abierta. Strawberry Canyon LLC. (Primera Edición, 1.0.5).

Modo Usuario

En modo usuario son accesibles un registro de dirección y configuración de PMP (Figura 0-6). El registro de dirección desplazado por 2 a la derecha, y si las direcciones físicas son menores que XLEN-2 bits de ancho, los bits más altos son ceros. Los campos R, W y X otorgan permisos de lectura, escritura y ejecución. El campo A establece el modo PMP, y el campo L bloquea el registro PMP y sus registros de dirección correspondientes.

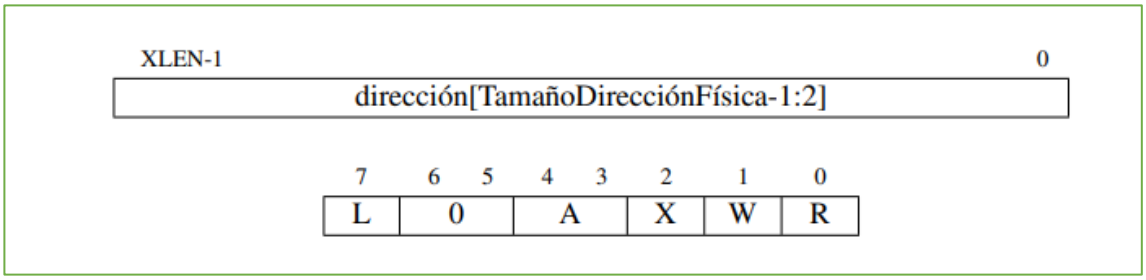

Figura 0-6. Registros de dirección y configuración PMP.
Fuente: Patterson, D. & Waterman, A. (11 de julio de 2018). Guía práctica de RISC-V: El Atlas de una Arquitectura Abierta. Strawberry Canyon LLC. (Primera Edición, 1.0.5).

Modo Supervisor

RISC-V dispone de un mecanismo por el cual las interrupciones y excepciones síncronas pueden ser delegadas al modo Supervisor selectivamente, evitando software de modo Máquina por completo, permitiendo reducir el tiempo que se invierte en manejar estas excepciones.

El CSR mideleg (Delegación de Interrupciones de Máquina) controla qué interrupciones son delegadas al modo Supervisor (Figura 0-7). Al igual que *mip* y *mie*, cada bit en mideleg corresponde al código de excepción del mismo número en la Figura 2-5. Por ejemplo, mideleg corresponde a la interrupción de temporizador de modo S; si está en 1, las interrupciones de temporizador de modo Supervisor transferirán el control al manejador de excepciones del modo Supervisor, en lugar del manejador de excepciones del modo Máquina.

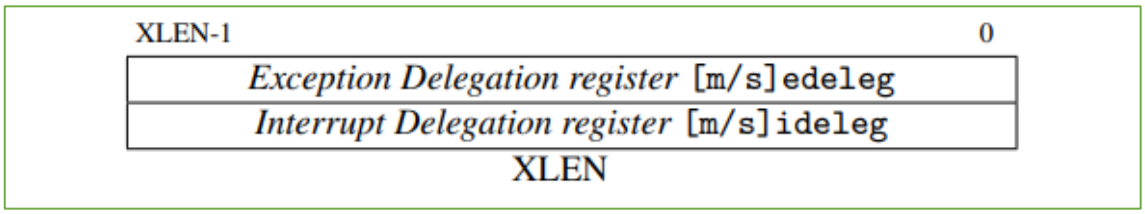

Figura 0-7. CSRs de delegación.
CSRs de delegación de excepciones e interrupciones de máquina y supervisor (medeleg, sedeleg, mideleg, sideleg).
Fuente: Patterson, D. & Waterman, A. (11 de julio de 2018). Guía práctica de RISC-V: El Atlas de una Arquitectura Abierta. Strawberry Canyon LLC. (Primera Edición, 1.0.5).

Cualquier interrupción delegada al modo Supervisor puede ser enmascarada por software en modo Supervisor. Los CSRs sie (Habilitador de Interrupciones de Supervisor) y sip (Interrupciones de Supervisor Pendientes) son CSRs de modo Supervisor y subconjuntos de los CSRs mie y mip (Figura 0-8). Únicamente en los bits correspondientes a interrupciones delegadas en mideleg es posible leer y escribir con sie y sip. Los bits correspondientes a interrupciones que no han sido delegadas siempre son cero.

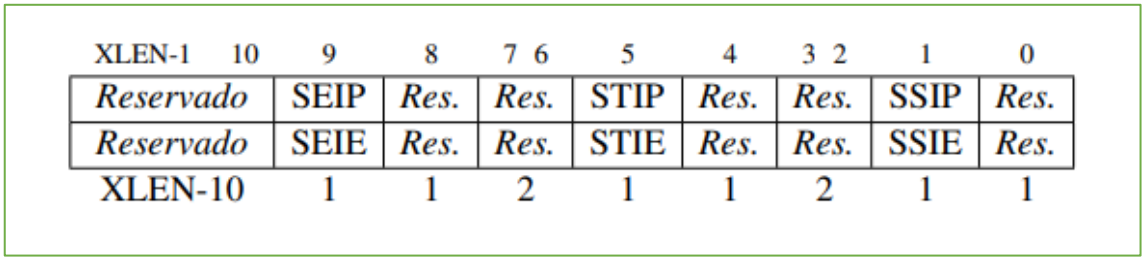

Figura 0-8. CSRs de interrupción de supervisor.
Son registros de XLEN bits de lectura y escritura que contienen las interrupciones pendientes (sip) y los bits de habilitación de interrupciones (sie).
Fuente: Patterson, D. & Waterman, A. (11 de julio de 2018). Guía práctica de RISC-V: El Atlas de una Arquitectura Abierta. Strawberry Canyon LLC. (Primera Edición, 1.0.5).

El modo Máquina también puede delegar excepciones síncronas al modo Supervisor usando el CSR medeleg (Delegación de Excepciones de Máquina) (Figura 0-7), pero las excepciones nunca transfieren el control a un modo menos privilegiado, es decir, una excepción que ocurre en modo Máquina siempre es manejada en modo Máquina. De la misma manera que una excepción que ocurre en modo Supervisor puede ser manejada por el modo Máquina o por el modo Supervisor, dependiendo de la configuración de delegación, pero nunca por el modo Usuario.

El modo Supervisor tiene varios CSRs de manejo de excepciones: scause, stvec, sepc, stval, sscratch y sstatus. La acción de tomar una excepción es muy parecida al modo Máquina. Si un hart toma una excepción y es delegada al modo Supervisor, el hardware pasa por varias transiciones de estados similares, usando CSRs de modo Supervisor:

- El PC de la instrucción que causó la excepción se guarda en sepc, y stvec se escribe al PC.
- scause recibe la causa de la excepción (Figura 2-5) y stval recibe la dirección defectuosa o alguna otra palabra con información de la excepción.
- Las interrupciones se deshabilitan escribiendo SIE=0 en el CSR sstatus y el valor previo de SIE se preserva en SPIE.
- El modo de privilegio pre-excepción se preserva en el campo SPP de sstatus y el modo de privilegio se cambia a S.

Para concluir, recordad que el modo supervisor está diseñado para soportar sistemas operativos modernos similares a Unix, tales como Linux, FreeBSD y Windows. El modo S es más privilegiado que el modo U, pero menos privilegiado que el modo M. Al igual que en el modo U, el software del modo S no puede usar CSRs ni instrucciones del modo M, y está sujeto a restricciones de PMP.